NAS2-11530

Report on an Evaluation Study of Data Flow Computation

George B. Adams III
Robert L. Brown
Peter J. Denning

Research Institute for Advanced Computer Science

RIACS TR 85.2
April 1985

RIACS

Research Institute for Advanced Computer Science

Report on an Evaluation Study

of Data Flow Computation

George B. Adams III
Robert L. Brown
Peter J. Denning

Research Institute for Advanced Computer Science

RIACS TR 85.2
April 1985

Prepared by RIACS under NASA Contract No. NAS 2-11530 and DARPA Contract No. BDM-S500-0X6000. The contents of this document do not represent the official position of NASA or DARPA.

EXECUTIVE SUMMARY

In September 1984 RIACS conducted a two-week study of the proposed MIT static data flow machine for applications of interest to NASA Ames and DARPA. NASA and RIACS scientists formed seven one- or two-person teams to study data flow concepts, the static data flow machine architecture, and the VAL language. Each team mapped its application onto the machine and coded it in VAL.

The application areas were computational fluid dynamics, computational chemistry, galactic simulation, linear systems, queueing network models, and artificial intelligence. The considerations for mapping these applications onto the machine were primarily architectural: the number of individual processing elements (PE), the size of the instruction memory in each PE, the speed of the PEs, the instruction issue rate, the size of the routing network among the PEs, and the size and speed of the array memory. The goal in mapping was to maximize the number of busy PEs and to minimize the traffic on the routing network. The target machine contained 256 PEs and was capable of an aggregate rate of 1.28 GFLOPS.

The principal findings of the study were:

1. Five of the seven applications used the full power of the target machine – they sustained rates of 1.28 GFLOPS. The galactic simulation and multigrid fluid flow teams found that a significantly smaller version of the machine (16 PEs) would suffice.
2. A number of machine design parameters including PE function unit numbers, array memory size and bandwidth, and routing network capability were found to be crucial for optimal machine performance. Thus, studies of this type can provide valuable feedback to machine architects.
3. The study participants readily acquired VAL programming skills. A very high level programming environment is essential to make the data flow machine usable by most programming scientists, however, because of the complexity of the machine architecture. For example, tools to aid debugging and mapping VAL programs onto the architecture are required.
4. We learned that application-based performance evaluation is a sound method of evaluating new computer architectures, even those that are not fully specified. During the course of the study we developed models for using computers to solve numerical problems and for evaluating new architectures. We feel these models form a fundamental basis for future evaluation studies.

PREFACE

This report describes a study held at RIACS during September 17-28, 1984. The study participants were:

George Adams, RIACS
Eric Barszcz, NASA Ames Research Center
Richard Briggs, RIACS
Robert Brown, RIACS
Peter Denning, RIACS
Scott Eberhardt, NASA Ames Research Center
Eugene Levin, RIACS
Marshall Merriam, NASA Ames Research Center
Harry Partridge, RIACS
Merrell Patrick, RIACS
Karl Rowley, NASA Ames Research Center
Catherine Schulbach, NASA Ames Research Center
Ken Sevcik, RIACS

Instruction in the data flow computer architecture and programming language used, as well as consultation and guidance, was provided by

Jack Dennis, MIT
William B. Ackerman, MIT
Gao Guang-Rong, MIT

Although Adams, Brown, and Denning took primary responsibility for preparing this report, all these people contributed significantly to the report. The individual team report summaries included herein were largely prepared by the team members.

TABLE OF CONTENTS

1. Introduction

During September 17-28, 1984, the Research Institute for Advance Computer Science (RIACS) conducted a data flow computation study. The purpose was to develop an assessment of the effectiveness of data flow programming using a specific data flow machine architecture for computational problems in several disciplines of interest to the sponsors. NASA Ames and DARPA. These areas were:

computational fluid dynamics
computational chemistry
galactic simulation
linear systems
artificial intelligence
queueing network models

Seven teams, each consisting of one or two scientists, studied data flow programming concepts, expressed important algorithms in the VAL programming language, and investigated how best to map their algorithms onto the MIT static data flow architecture. Three researchers from MIT (Jack Dennis, William Ackerman, and Gao Guang-Rong) served as teachers and consultants. RIACS provided the offices and computing equipment, planned and organized the schedule, collected the data and results, and produced this report.

This report contains an overview of the study: what took place and what we learned. Included is a summary of the individual projects and their conclusions. The reader is advised to read carefully and critically, and add his own conclusions to those we present.

Because of the two-week time constraint, we were unable to do several things that might have improved the evaluation of the static data flow machine. We were unable to run complete VAL programs through the interpreter because the algorithms and programs used by the teams were too complicated to code to completion during the workshop. Because of this, we could not directly compare the cost and time to run the VAL versions to the FORTRAN versions run on the Cray X-MP. We believe these comparisons are worthwhile; they are left for future study.

2. Background and Motivation

2.1. Matching Computational Models and Problems

The computational needs of science and engineering have reached the limits of single-processor supercomputer technology. It is unlikely that by 1990 there will exist a single-CPU computer capable of more than 2 or 3 GFLOPS (Giga Floating Point Operations Per Second) but routine problems in science and engineering will demand computers 10 or 100 times that fast.[1] See [Adam84]. The required computational power is attainable only with computing systems consisting of many machines executing simultaneously on a different parts of the solution. Here "many" means hundreds or thousands of machines. Such systems are called "concurrent processing systems." (VLSI circuit technology complements this direction in computer architecture.)

Computer scientists have traditionally studied "models of computation," ranging from the inherently sequential, as with the common program-counter based model, to inherently parallel, as with data flow. Each of these models may have one or more abstract machines that implement them and these abstract machines may in turn be realized by concrete machines. A popular model of parallel computation frequently realized by a large set of sequential machines communicating over a network. This adds a new level of complexity to programming which now must explicitly consider the communication and synchronization among many parallel activities.

Now: For any given class of problems there may exist several qualitatively different practical, parallel algorithms that solve problems from that class. Some of the models (hence, machines and associated languages) will be well suited for a given algorithm, others not. Thus, in the world of concurrent computation, it becomes interesting -- and important -- to ask, "Which combinations of problem-domain and models (computation and communication) are most effective?" The answers will be based on evaluations of two kinds:

1. Objective assessments such as program size, running time, and cost per solution. These assessments can be made partly by mathematical analysis and partly by experiment.
2. Subjective assessments of human factors such as programming time, ease of finding good solutions using the given architecture, understandability of programs, and quality of the programming environment. These assessments must be made by experiment.

Evidently, a large effort would be required to systematically compare architectures among various disciplines in order to answer the question. How to organize such an effort is the subject of another report [Adam85].

[1] Because problem-time is often a polynomial (or worse) function of problem-size, it is inevitable that computational needs will surpass the power of sequential machines. For example, if the best algorithm for a problem takes time n^2, a double-speed CPU could handle a problem only about 40% (factor of $\sqrt{2}$) larger in the same amount of time.

The RIACS data flow computation study was a first attempt to answer the question for a specific machine (the MIT static data flow machine) and selected disciplines. The two specific purposes were:

1. To obtain a preliminary answer to the question, "How effective is data flow computation, as realized in the MIT VAL language and static data flow machine, at solving problems in disciplines where NASA and DARPA are seeking breakthroughs in computational power?"
2. To experiment with a prototype for studies that identify the most effective combinations of domain and models.

The results of this study are not a final answer to the question posed in (1) immediately above. The reasons include the following:

1. Since no one has prior experience with such studies, we could only make educated guesses at the methodology that would allow data flow computation to be compared across a variety of dissimilar disciplines.
2. Not all disciplines of interest to NASA and DARPA were covered because the release time for scientists participating in the study was difficult to negotiate.
3. Only a handful of relatively simple algorithms for solving "kernel problems" of the participating disciplines could be addressed in the short time available (two weeks). One must use caution in extrapolating the results to larger problems, full systems, and complete disciplines.

2.2. Approaches to Evaluating Machine Performance

There are four levels at which to evaluate the performance of any computing system. They differ in their requirements on the amount of support required to program them and in the knowledge required to perform the evaluation. The four levels are as follows:

1. **Raw performance evaluation.** The burst performance of a machine, expressed in operations per second, may be computed from knowledge of the clock speeds, memory and register speeds, and bus speeds. This provides an absolute upper bound on the speed of any computation performed on a machine, but achieving it in any real computation is a practical impossibility. No programs need be written to evaluate a machine at this level.
2. **Small function programming.** Small common functions, such as matrix multiply or FFT, are meticulously coded, usually in machine or assembly language, to achieve maximum speed. The functions are run on the machine and tuned. The only programming support needed is a compiler or an assembler.
3. **Benchmark programming.** An existing software package, such as LINPACK, or a software standard, such as Livermore Loops, is programmed for the machine. It is run and timed for a particular input. Programming

support requires compilers, linkers, and an operating system.

4. **Complete application programming.** The machine is evaluated by programming and observing an entire and real application on it. The programming is usually best performed by experts in the domain of the application. An entire programming environment consisting of editors, compilers, linkers. and debuggers is required.

The first level is cheap, but crude, and gives no information about the machine's programmability. The second level gives good ways of "stress-testing" to discover the fraction of instantaneous rate delivered to tightly-coded routines. The third level provides a basis for comparing the new machine to existing machines running the same standard software. The fourth level gives an assessment of the machine's programmability and its applicability to a particular domain.

Our goal in this study was to shed light on the "best-match" question described earlier. Because this inherently includes the question about the programmability of the machine, we were obliged to undertake the evaluation at the fourth level.

3. Problem Solving Process

Because this study focuses on the ability of a machine to support problem-solving in given disciplines, it is necessary to consider explicitly the process one uses to solve a problem computationally. The problems considered by our teams had been previously solved by FORTRAN programs running on DEC VAXes or Cray computers. Using the model for problem solving, we hoped to determine the best place to deviate from the previous solution path to the optimal solution path targeted for the MIT static data flow machine. It should be noted that this model is not general-purpose, but it describes the process used in the disciplines in our study fairly well.

Our model says that the problem-solving process consists of a sequence of increasingly detailed solution representations, the transformations between them, and the knowledge required to perform those transformations. The sequence of representations culminates in a computer program. Initially, there exists a (prose) problem statement. We call this *stage zero*. Using knowledge of his discipline, the scientist first states a solution to the problem using the language of mathematics, usually continuous mathematics. This solution may be a simple function or a complicated sequence of equations. We call this *stage one*.

The next step is to transform the solution mathematics into an abstract (machine independent) algorithm. If there is no direct way to solve the problem using the mathematics, this is the stage at which the approach to the numerical simulation is determined. This transformation introduces discrete approximations to continuous mathematics, numerical methodologies, algorithmic approaches, error controls, recursions, and partitions into potentially parallel components. The knowledge used includes those techniques, and knowledge of the particular model of computation implemented by the target machine, though not necessarily detailed architectural parameters. The language used here is a combination of discrete mathematics and a high-level pseudo language. We call this *stage two*.

The third step is to render the abstract algorithm as a program in a computer language. This transformation includes decisions about data representations and control flow. The knowledge needed includes the definition of the computer language used and the architectural parameters of the machine. We call this *stage three*.

The final step is to translate the program of stage three into a machine code. This step is usually performed by a compiler embodying knowledge of the instruction set and data representations of the machine.

In the context of our study, the sequence of stages is depicted by Figure 1. When employing a new machine of different architecture and model of computation from the current machine, it is best to backtrack to stage one and formulate a new abstract algorithm better suited to the new model of computation. If, however, the new machine is not based a new model of computation, as is the

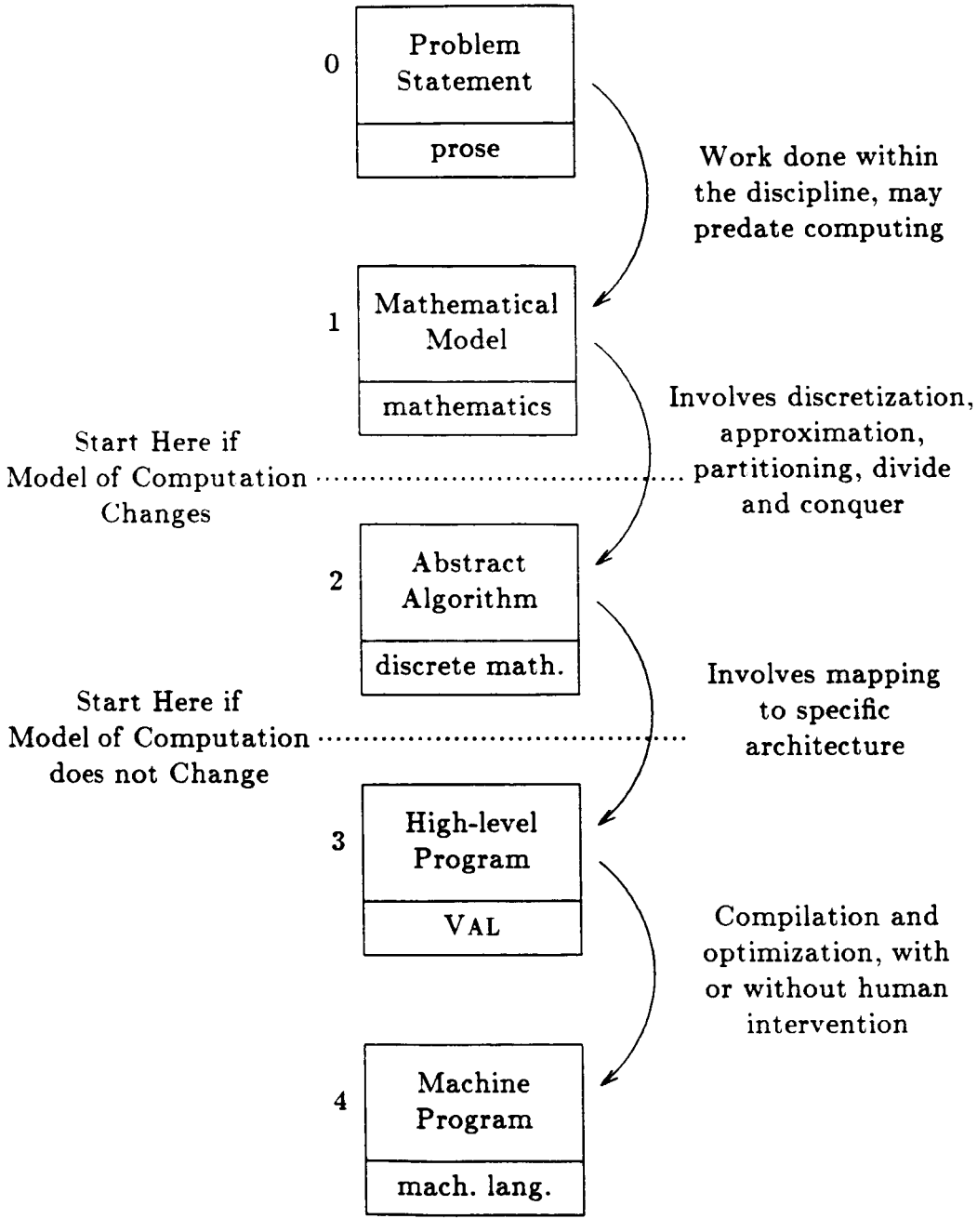


Figure 1. Steps in Problem Solving

case when moving an application from one vector processor to another, backing up to stage two may be adequate.

We observed that in our study, most participants backtracked to stage two as given. They reported that the abstract algorithms for their problems were already in a highly parallel form, though it required thought and examination over several days to discover this. That five of the seven teams then constructed codes that ran at the full speed of the machine tended to confirm their beliefs. However, in future studies, it would be well to require teams to provide explicit arguments why a further retreat to stage one would produce no significant benefit. Existing algorithms for a given application that readily allow even a great deal of parallelism cannot be assumed *prima facie* to be high performance.

4. Methodology

The study participants were organized into teams. Team members were experts in their respective disciplines and knowledgeable in the use of supercomputers for the solution of problems in that discipline. The team members and their disciplines are given in Table 1.

Team members were volunteers who were personally interested in the study. They were chosen to constitute a cross-section of the research areas of interest to the sponsors. Obtaining release time for some researchers was difficult; two weeks was the maximum time for which most could be available.

Table 1. Study teams.

Discipline	Team Members
Computational Fluid Dynamics I	Scott Eberhardt Karl Rowley
Computational Fluid Dynamics II	Marshall Merriam
Computational Chemistry	Harry Partridge Eugene Levin
Galactic Simulation*	Eric Barszcz Cathy Schulbach
Linear Systems	Merrell Patrick
Artificial Intelligence	Rick Briggs
Queueing Networks	Ken Sevcik Peter Denning

* This team comprised proxies for the original algorithm designers, who were unable to be present personally.

Team selection was complete-one month prior to the study. At that time, team members were given some general information on data flow that they could read at their convenience. There was no formal instruction in data flow prior to the study, nor had any of the participants received any. All team members were, however, previously aware of data flow computation.

Each team brought a working program or detailed algorithm specification from its application domain to the study. The study began with intensive instruction in data flow concepts and the programming language VAL conducted by the MIT contingent. Thereafter the teams sought to apply data flow methodology to their algorithms. Each team, with assistance as needed from MIT personnel, programmed and transformed their application for execution on a data flow computer.

The first four days of the two-week study consisted of half a day of lecture and discussion and half a day of team study and programming. The remaining six days were team study and programming, with occasional reports by team members to the entire group. The lectures early in the workshop concentrated on the VAL language and the architecture of the MIT static data flow machine as proposed for this study. During these early days, the participants wrote sample programs to familiarize themselves with the VAL language and the RIACS computing system. We anticipated that it would take one full afternoon for participants to familiarize themselves with the system; it took about one hour. We anticipated it would take one week for them to learn VAL; it took three days. The steps they carried out were:

1. Characterize their application in terms of computational blocks and flow of data among blocks. (This is called "pipe-structured data flow methodology.")
2. Select representative and critical blocks for coding in VAL.
3. Design the structured data flow machine code.
4. Evaluate the performance the specified static data flow computer would provide.
5. Iterate the above steps (time permitting) to obtain improved solutions.

The participants used the RIACS computer facility to conduct this work.

Arrangements were made to provide office space for exclusive use by the study. This also removed the participants from their daily milieu of telephone calls and other distractions and increased the amount and quality of time the participants devoted to the study. By providing emulators, we attempted to minimize necessary operating system interaction and to simulate the command level interface of the editors familiar to the participants. In this way team members spent the majority of their time on data flow investigation, rather than on learning a new operating system and editor.

During the work/programming hours, the MIT researchers worked closely with the teams, helping them map their codes onto the data flow machine. Throughout the study, RIACS personnel helped with problems related to the facility, and collected data concerning the reaction of the team members to data flow and their success at using the language.

Each team prepared a report on its work during the study. These reports are summarized in the following sections overviewing the team projects. The full reports are to be available as separate RIACS and NASA technical reports. In addition, we handed out a questionnaire aimed at determining the participants' reactions to data flow programming and the format of the workshop; a summary appears in an appendix.

During the study, the VAL translator was used approximately 500 times. We captured most (62%) programs that passed through the translator (some were not because we assumed participants would consistently use file names of the form *name.val* for all VAL programs; some did not). All interpreter files, interpreter dialogs, and compiler error messages were captured. This allows us to "replay" any captured portion of the study. Participants were not informed that programs were being captured so that their use of the VAL system would not be biased.

5. The MIT Static Data Flow System

[Editors' note: The technical specification of the MIT static data flow machine and VAL was provided by the MIT team].

The description of the data flow system has four parts: the model of computation, the machine architecture to implement that model, the language VAL used to program the machine, and the optimizing compiler for VAL.

5.1. Model of Computation

The model of computation implemented by data flow is significantly different from the more familiar control flow model used in traditional Von Neumann machines. This section describes the basic concepts: for a thorough overview, see [Davi82].

A data flow computation is represented as a directed graph, each node representing a single operation and having one or two incoming edges and a single outgoing edge. Conceptually, an outgoing edge may be split and become an incoming edge for several other nodes. Data values move as tokens on the edges of the graph. When a node has a token on all of its incoming edges, it becomes enabled to perform its computation, or "fire," consuming the incoming tokens and generating a result token on the outgoing edge. Generally, firing rules include the condition that there be no token on the outgoing edge.

Loops can be created by making the graph cyclic, and imposing initial conditions on some edges. Additionally, conditionals can be created by using special nodes with an added Boolean input. One type of conditional node passes one of its two inputs to its output depending on the state of the Boolean input. The other type of conditional node passes its sole input to one of two output depending on the flag.

Large programs can be composed of smaller graphs, and parallelism can be achieved because of the firing rules. Since the edges on the graph represent the data dependencies among the operations, any node with all its inputs satisfied may be fired, regardless of whether other nodes around it are currently firing. Likewise, data may be pipelined through a data flow graph, with the initial nodes in the graph consuming input in streams [Gao82].

5.2. The Machine

Many of the design parameters of the machine under study are not firm. What follows is a description of the machine as presented at our study by the MIT group and used by the participating scientists. During the course of the two weeks, some of the parameters changed slightly, and they may continue to change in the future, perhaps as a result of this study. Other designs for data flow computers exist, for example [Gurd85, Rumb77].

The data flow supercomputer suggested as the target for the performance study consists of 256 processor elements interconnected through a routing

network. Each processor is capable of executing any data flow element, and generally the whole program is partitioned and spread across all processors in a way that preserves locality. Under the best conditions, the result of an operation will be only needed in the same processor in which it was computed, however, if it is needed in another processor, it will be sent through the network. Each processor has a separate instruction store and array memory. I/O devices and mass storage are attached to the system through the network. A block diagram of the architecture is shown in Figure 2.

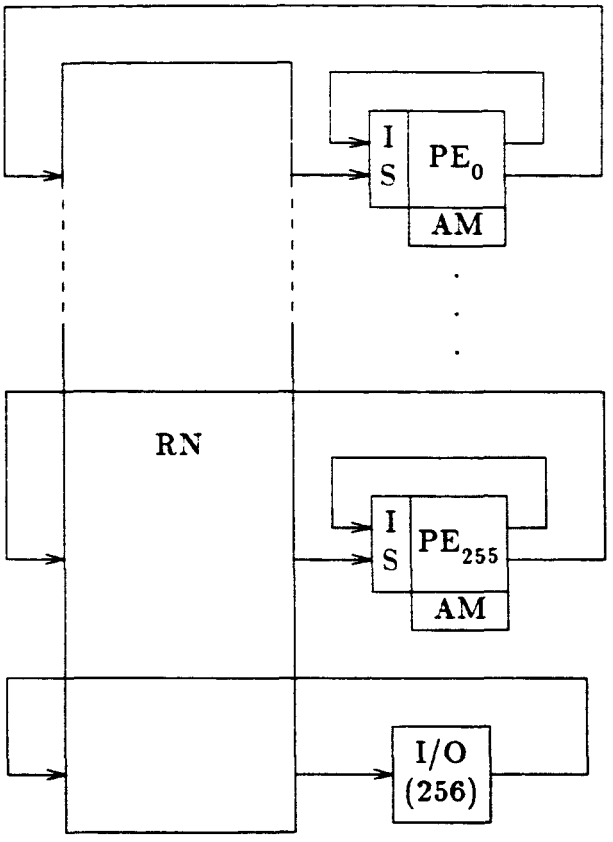
Each processing element (PE) is designed around the Weitek 64-bit floating point chip set. It includes one adder chip capable of 5 to 8 MFLOPS performance and two multiplier chips capable of 1.25 to 2.0 MFLOPS performance apiece. The instruction cell memory of a PE is divided into two regions -- 1024 cells for floating point operations (primarily adds, subtracts, and multiplies), and 1024 cells for "red tape" instructions (the integer arithmetic, buffer manipulation, tests, etc., needed to control the floating point computation). The idea is that each PE should easily hold enough red tape instructions to ensure that the floating point chips are kept busy.

Assuming that a floating point (FP) operation is begun every 200 nanoseconds, and it takes at most 2 microseconds to completely process an enabled instruction (from setting its enable flag to setting the enable flags of target instructions), ten active FP instructions are sufficient for the PE to run at peak FP rate. If a pipelined section of machine code is spread over many PEs, and the pipeline headway is 50 microseconds, then each PE will have to hold 250 FP instructions of the pipeline to operate at peak performance.

There are several reasons that more than 250 FP cells should be provided: (1) Due to conditional computations there will be some FP cells that are not used on every pipeline cycle. (2) Some computations will run in several phases, and, to achieve full performance, each phase separately will have to fully utilize every PE; hence each PE will hold code for each phase of the computation being performed. (3) Miscellaneous instructions will be needed for initialization, input/output, and for other functions peripheral to the main computation. It may turn out that the total of 2048 total instruction cells is too low.

The square root and divide operations (actually reciprocal square root and reciprocal, respectively) for floating point values will be supported by performing Newton iterations from an initial guess obtained from a ROM.

The routing network (RN) has an input and an output port for each of the 256 PEs and and additional set of 256 input and output ports for mass memory devices, display systems, and the host processor. The RN will have nine stages with 256 two-by-two routers in each stage, for a total of 2304 router modules. The links between routers have a 16-bit data path and can operate at 5 MHz or better. A typical result packet sent through the network consists of an 8-byte FP value and four bytes for target PE and instruction cell identification, etc. Thus a router will accept a packet in 1.2 microseconds. Because of contention,



RN Routing Network. 512 by 512, 16 bit data paths, operates at > 5MHz, average rate of transmitting FP packets 0.25 MHz from a single PE to another.

PE Processing Elements. 5 to 8 MFLOPS with 1.25 to 2 MFLOP multiplies. 256 PEs in the system.

IS Instruction Store. 1024 cells for FP instructions, 1024 for others.

AM Array Memory. Size not fully determined. At least 256K 64 bit words per PE.

I/O Input/Output. Includes mass memory, host processor, display systems. 256 ports to the RN are reserved for I/O.

Figure 2. Static Data Flow Machine Architecture

we assume that the network operates from PE to PE at 30 percent its maximum rate (this derating assumption may be subject to significant change once actual

experience is gained), yielding a communication rate of 0.25 MHz. A simulation study of the Delta network (topologically equivalent to the RN) shows the need for derating performance [Dias81].

The amount of array memory to be attached to each processing element is a matter of debate. For the study 256K words of 64 bits for each PE, making a total of 60M words in the machine, is assumed. The transfer rate for this memory is 2 MHz or better, and the latency for reading is about two microseconds. The desirable performance of the array memory is also controversial.

The Array Memory is supplemented by a Disk Storage System (alternatively, a solid state mass memory) that communicates with the PEs through the Routing Network. The performance to be expected from the disk system is about one megabyte per second transfer rate for each disk unit of which there might be 32 or 64, for example. This yields a total of 4 or 8 million FP values read or written each second. The disk capacity available could be huge, but there is little hope of exchanging that excess capacity for greater transfer rates without expensive redesign of the disk units.

5.3. The Language

The MIT team has developed the language, VAL, for the data flow machine. This language is described in [Acke82] and documented in [Acke79].

The language is functional in nature and is value-oriented. It is similar to most modern languages in that all the common scalar data types are supported, as are arrays and aggregate types. The primary difference is that all data are treated as values, not objects. In an object-oriented programming language, computations take the form of operations on objects, either to extract information or to change the state of an object in a controlled, well-defined way. In VAL, however, there are no data objects, only values. Hence, it is not possible to make elemental changes to arrays, for example. Instead, new arrays are made of old arrays by changing elements.

Programs are constructed of functions in VAL, and each function may compute one or more values. No static storage is permitted within functions, as the data flow model of computation itself has no concept of static storage.

Two control structures within VAL deserve special mention. These are **for-iter** and **forall-construct**. Both of these control structures define program loops. Program loops for iteration have four basic parts: (1) definition of initial values of control variables, (2) loops termination test, (3) computation performed by the loop, and (4) modification of loop control variables for next iteration. The **for-iter** construct in VAL makes the four parts distinct because (4) invariably takes the form of assignment statements to replace the old values of the loop control variables with new values, a side effect that is not normally permitted in the language. This case denotes an inter-iteration dependency that usually cannot be computed in parallel. However, depending on the nature of

the computation, the compiler may try to discover potential parallelism in (3), the loop computation.

Program loops are also used to create arrays. In this case, there often are no inter-iteration dependencies, and each iteration may be computed in parallel with the others. For this reason, VAL has the **forall-construct**, which allows the programmer to define the construction of one or more arrays. An example of the use of **forall-construct** is as follows:

```
X, Y :=
  forall I in [1, N]
     A : real := f(I)
     construct I, A
  endall
```

In this case, the machine can compute all N values of the two arrays in parallel, and the results will be available as the values X and Y.

5.4. The Compiler

The MIT compiler for VAL generates an intermediate language and will be targeted for the static data flow machine.

Optimizations and mapping decisions made by the compiler are of critical importance [Acke84]. The compiler that MIT intends to construct will perform at least the transformations given below. Until a great deal of additional experience is obtained, they expect that the decisions about optional or "parameterizable" transformations will not be made automatically. They will be made under the control of information provided by a human. This will take the form of an "advice file" associated with each program. Once the advice is given, the indicated transformations will be made automatically.

Of the transformations listed below, the most important are loop unfolding, array interlace, streaming, and pipelining.

Small Array and Record Removal. Small arrays (say 10 elements or less) are often best handled by being broken up into separate tokens containing the individual values. Small **forall** loops creating such arrays will be similarly expanded. For example

A := **forall** I in [1, 3] **construct** f(I) **endall** ;

will become separate subgraphs to compute f(1), f(2), and f(3), which will be passed through the graph on separate edges. References to A[2], for example, will then become trivial.

Records will be treated as small arrays and handled similarly.

Array and Loop Transposition. If we have a large forall inside a small one, as in

A := **forall I** in [1, 3]

construct forall J **in** [1, 10000] **construct** f(I, J) **endall**
endall ;

the outer **forall** and outer level of the array will be removed as described above, resulting in three separate foralls creating three separate one-dimensional arrays.

If they are in opposite order, as in

A := **forall** J **in** [1, 10000]
construct forall I **in** [1, 3] **construct** f(I, J) **endall**
endall ;

the loops will be "transposed", that is, their nesting will be reversed. All subsequent references to the array A will be likewise transposed, turning A[P+Q,3] into A[3,P+Q]. The removal of the small array and small **forall** can then proceed. This transformation is important in such things as block-tridiagonal matrix processing.

Loop Unfolding. In any repetitive calculation extra parallelism among the cycles will be exploited by evaluating many cycles at once in different parts of the data flow graph. For example, if a loop is unfolded by a factor of 8, there will be 8 separate loops in the data flow graph, all running approximately in step with each other. The first piece will evaluate the first cycle of the original loop, then the 9th, then the 17th, etc. The second piece will evaluate the 2nd cycle, then the 10th, and so on.

How much unfolding to perform, and how to map the separate pieces onto the different processing elements of the computer are decisions that will come from the advice file. Often, the number of pieces will be greater than the number of processing elements, so each processing element will contain several pieces.

Array Interlacing. When a loop that accesses an array sequentially is unfolded, it will be appropriate to separate the array into a number of pieces, with each piece of the loop accessing its own private piece of the array. This reduces inter-processor communication and increases the effective bandwidth of the memory system.

The division of the array will usually take the form of interlacing, with the first piece holding, in consecutive locations, A[1], A[9], A[17], etc, the second holding A[2], A[10], A[18], and so on.

Streaming and Pipelining. A great many loops that manipulate arrays do so by performing the same computation repeatedly on each element of the array. When this occurs, the array is processed as a stream of scalar values passing through the data flow graph in sequence. Different parts of the program can pass arrays to each other in this form, without storing them in memory during intermediate stages of the computation.

When an array is too large to fit in the RAM and must be stored on the disk, it will be transferred to and from the data flow graph in the form of a stream.

Loops that process arrays as streams of tokens will be pipelined [Denn83]. The graph that comprises the body of the loop will have many consecutive "waves" of tokens inside it in various stages of progress. This is the principal method by which the percentage of operators that are enabled is increased to improve utilization of the processing units.

6. Overview of Individual Projects

What follows are summaries of the individual team reports prepared largely by the teams themselves. They have been edited as necessary to conform to a common style and meet space constraints. Most of these summaries are distilled from full-length reports prepared for publication by the team members. The form of each summary follows that presented in the section on problem solving techniques, with the addition that some background information is given in the "Problem Overview" section. For reports on previous studies on mapping applications on the MIT static data flow machine, see [Denn84a, Denn84b].

6.1. Computational Fluid Dynamics - 1

Team: Scott Eberhardt & Karl Rowley

Problem Overview. The computation under study is the Conservative Supra-Characteristic Method (CSCM) for numerically solving the equations of fluid motion: the Navier-Stokes equations for viscous fluids, or the Euler equations if viscosity is neglected. The author concentrated on the one-dimensional model, but developed strategies for programming the three dimensional model.

Mathematical Model. The model used for this study is the time-accurate Euler equations. These equations describe the conservation of mass, momentum, and energy. For the one dimensional case, they are

$$\frac{\partial q}{\partial t}+\frac{\partial F}{\partial x}=0$$

where

$$q=\begin{bmatrix}\rho\\ \rho u\\ e\end{bmatrix},\quad F=\begin{bmatrix}\rho u\\ \rho u^2+p\\ u\left(e+p\right)\end{bmatrix}$$

and

$$p=\left(\gamma-1\right)\left[e-\frac{1}{2}\rho u^2\right]$$

The elements of the vector q are called the conservative variables and the vector F is the flux vector. The variables are ρ= density, u = velocity, e = total energy per unit volume, and p = pressure.

Abstract Algorithm and Approach. The detailed analysis and development of the algorithm to solve for the conservative variables in the above equation is too complicated for inclusion in this report. The interested reader is referred to [Eber85].

Many CFD algorithms use finite differencing, where partial derivatives are represented by finite differences. The domain is first broken into a discrete system and differences are computed between the discrete points. Most algorithms

for solving the Euler equations are implicit, and hence represent the solution at each point in the system as a function of other points in the system. Therefore, the solution procedure will involve the inversion of matrices which are usually block tridiagonal, scalar tridiagonal, scalar pentadiagonal, or, occasionally, block bidiagonal. The block matrices result from the system of equations so that the one dimensional problem, with three equations, produces 3×3 blocks. The Euler equations basically represent a convection process while the Navier-Stokes equations also include a diffusive process. The convection part of the two equation sets give rise to a wave property which can be exploited in the algorithm. CSCM decouples the system of equations into three wave equations and combines positive waves together in one set of equations and negative waves in another set. The equations are then finite-differenced in such a way as to capture the correct wave propagation direction. The equations are then recombined to form the complete system.

The resulting algorithm used in CSCM is written as follows:

$$\left(I + \tilde{A}_{j-1}^{+} \nabla_x + \tilde{A}_j^{-} \Delta_x \right) \Delta q = -\left(\tilde{A}^{+} \nabla_x q^n \right)_{j+1} - \left(\tilde{A}^{-} \Delta_x q^n \right)_j$$

where

$$\nabla_x f = \frac{1}{\Delta x} \left(f_{j+1} - f_j \right)$$

$$\Delta_x f = \frac{1}{\Delta x} \left(f_j - f_{j-1} \right)$$

When taken out of operator form, the lefthand side becomes

$$-\left[\frac{\tilde{A}_{j-1}^{+}}{\Delta x} \right] \left(\Delta q \right)_{j-1}^{n+1} + \left[I - \frac{\tilde{A}_{j-1}^{+}}{\Delta x} + \frac{\tilde{A}_j^{-}}{\Delta} x \right] \left(\Delta q \right)_j^{n+1} + \left[\frac{\tilde{A}_j^{-}}{\Delta x} \right] \left(\Delta q \right)_{j+1}^{n+1}$$

or

$$-A \left(\Delta q \right)_{j-1}^{n+1} + B \left(\Delta q \right)_j^{n+1} + C \left(\Delta q \right)_{j-1}^{n+1}$$

The indices j and n represent spatial and time discretization, respectively. Each step of the algorithm computes a new Δq_j; this being used to compute the new values of the conservative variables at each grid point j.

The matrix $\tilde{A}$ transforms the conservative into the right- and left-running waves, hence the existence of $\tilde{A}^{+}$ and $\tilde{A}^{-}$.

When extended to multiple dimensions, the dimensions are decoupled to look like multiple one dimensional problems.

The Program. The overall structure of the program is shown in Figure 3.

There are two ways in which the one-dimensional CSCM code can be implemented. The example code is small and so will not utilize the full machine in either implementation but the analysis will lead to a more clear picture of the multi-dimensional problems which will be covered later. The first method is to

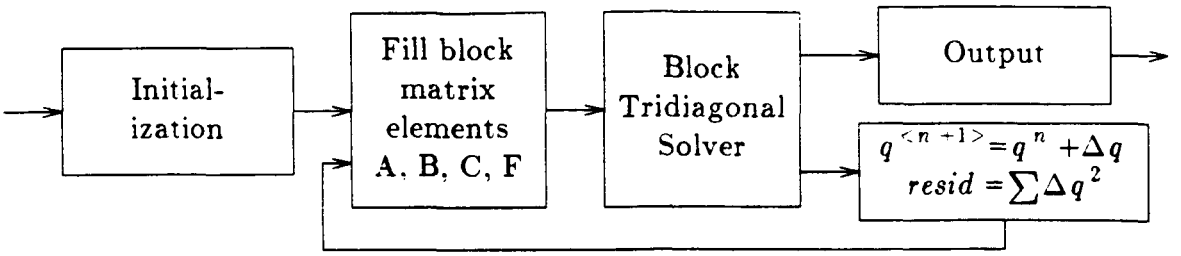


Figure 3. Structure of CSCM Data Flow Program.

pipeline the code on a single processing element (or through a few processing elements), leaving all other processors idle. The total number of operations required for the multi-dimensional CSCM code is greater than the instruction cell memory will allow so more than one processor must be utilized for the pipeline chain. There are two pipelines in the 1-D code represented by the block which fills the block tridiagonal elements and the block tridiagonal inversion routine. The scientist concentrated on the filling part because a previous study examined the matrix inversion block. Analysis showed that the pipeline length for this step is 368 floating point operations, exceeding the required 250 operations to keep the pipe busy. Hence, the one-dimensional code can adequately keep one processing element busy. The other technique for solving the one-dimensional case is to spread the computation over all 256 processors. However, rarely is the one-dimensional case computed for so many grid points and the interprocess communication would overload the routing network.

For the multidimensional case, the grid space is broken into "pencils," where each pencil is a one-dimensional cut or line where all coordinate values except one stay constant. The method of approximate factorization was used to decouple coordinate operators into pencils. A four step process is used to obtain Δq. The first is to compute the right-hand side operator which is a function of known variables. The next three steps are to compute the block tridiagonal elements and invert the block tridiagonal matrices for each of the three coordinate directions. The relation to the one-dimensional problem results from these three sweeps.

Conclusions. As this algorithm is very similar to the one used by the MIT team during the early design phases of the machine, it maps well onto the static data flow architecture. It is estimated that a sustained performance of 1 GFLOP is achievable. Shortcomings in the architecture include the small instruction storage. If the program were modified to allow each PE to process

an entire X-Y plane, the array memory requirements for the plane alone would expand to 150K words.

6.2. Computational Fluid Dynamics - 2

Team: Marshall Merriam

Problem Overview. A multigrid Euler equation formulation for computational fluid dynamics (CFD) is a simplified model of fluid flow, neglecting viscous effects. but one that is commonly used in the aerospace field and of great importance. For the data flow study a multigrid solver for Euler equations (FLO52R) was examined to determine its performance on the proposed MIT static data flow computer. This code is competitive with the fastest Euler solvers available for the Cray X-MP and is widely used in industry. For a detailed report of this team's study, see [Merr85].

Abstract Algorithm and Approach. The program FLO52R was used as the starting point for the study. Although it is explicit for the most part, there is a small implicit section to do smoothing on the residuals. The scientist quickly concluded that the code in its existing state was a bad match to the architecture. First. the code contains a subroutine which solves a number of scalar tridiagonals in each spatial direction. This results directly in limited parallelism due to a data dependency of the first-order recurrence type. Second, a significant part of the algorithm involves solutions of the Euler equations on coarse grids. Since parallelism is limited by the number of mesh points, even the explicit portion of the code becomes a potential bottleneck. Additionally, multigrid codes are more readable if they employ recursion when changing grid meshes, but recursion is disallowed in VAL.

FLO52R utilizes a full multigrid sawtooth cycle, solving the equations on an initial grid, restricting the grid to a coarser grid, solving again, interpolating back to the finer grid and solving once more. This cycle repeats. employing an increasingly wider range of grids. In particular, FLO52R cycles between 32×8 and 64×16 point grids 40 times, then between 32×8, 64×16, and 128×32 point grids 40 times, then four grids (adding a 256×64 grid) 400 times.

The Program. For this study. FLO52R was viewed as one pipeline, many instructions long and many instructions wide. The scientist did not perform a complete coding of the application, but rather focussed on some of the key parts of the computation. The remainer of this subsection displays one such key part. The interested reader is referred to the longer report [Merr85] for more detail on this and other portions of the computation.

As an example of how an explicit portion of the code might be programmed, Figure 4 shows the FORTRAN code and corresponding data flow graph for the second order smoothing subroutine. For clarity, the loop for only one direction, one variable, and one row of data is shown. In practice all four variables, both directions, and 64 rows of data could be executing simultaneously with a

```
      SUBROUTINE FILTC(I2.W.P.VOL.DTL)
C FOUR EQUATION MODEL
C SECOND DIFFERENCES WITH FIXED COEFFICIENT
      COMMON /C/ IL. IJ
      COMMON /E/ VIS0
      COMMON /FIL/ FW(257.65). RFIL
      DIMENSION W(I2.1). P(I2.1). VOL(I2.1). DTL(I2.1)
      DIMENSION FS4(257)

      FIS0 = VIS0/32.

      DO 20 J=2. JL
            DO 10 I=1. IL
                  FIL = FIS0 * (VOL(I+1.J)/DTL(I+1.J)+VOL(I.J)/DTL(I.J))
                  FS4(I) = FIL * (W(I+1.J)-W(I.J)+P(I+1.J)-P(I.J))
10          CONTINUE
            DO 20 I=2.IL
                  FW(I.J) = (1. - RFIL)*FW(I.J)+RFIL*FS4(I-1)-FS4(I))
20    CONTINUE
      RETURN
      END
```

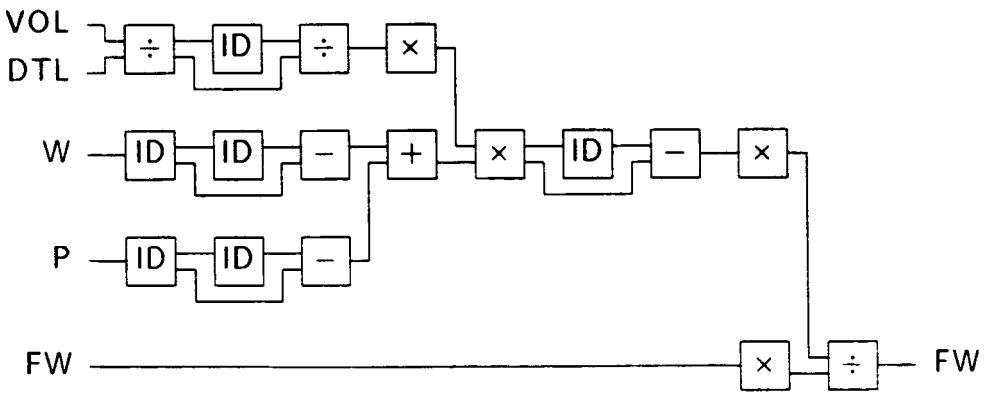


Figure 4. FORTRAN Code and Data Flow Graph, Explicit Smoothing

transpose between the two directions. The boxes labeled "ID" are "identity" instructions and are used, for example, in the second row of the data flow graph to implement the W(I+1.J)-W(I.J) expression.

In the figure, the input data are shown on the left. They arrive as an ordered stream with W(1) arriving before W(2) which arrives before W(3). Notice how the averaging and difference operators are implemented by taking inputs from different points in the input stream. The output, FW, appears as an ordered stream on the right.

Other sections of the longer report discuss the handling of boundary conditions, transposes, and the implicit second order smoothing of the residuals.

Conclusions. Several assumptions were made concerning performance of the machine and the nature of the activity that would occur during the course of the calculation. One was that all network traffic consists of floating point data traveling between subroutines. It was found that for the FLO52R code there were three bottlenecks that prevent effective use of all 256 PEs in the proposed architecture despite the explicit algorithm used. The network used must be able to support matrix transposition if performance is not to suffer. In spite of this, performance levels approaching that of a Cray-1 may be possible for a computer costing far less.

The problems with the data flow machine are its tiny scalar speed, need for massive temporary storage, prohibition of recursion, and lack of debugging support.

6.3. Computational Chemistry

Team: Harry Partridge & Gene Levin

Problem Overview. In quantum chemistry we determine the properties of atoms and molecules from first principles by solving the time independent Schrodinger equation. The solution algorithm we employ involves a double basis set expansion of the wave function Ψ using a variational principle or a perturbation expansion to optimize the parameters. The quantum chemistry techniques are capable of providing accurate atomic and molecular properties such as molecular geometries, dissociation energies and transition probabilities. The calculated properties both complement and supplement the available experimental data. In addition, the results can provide qualitative insight of chemical phenomena. The steps selected for study in the workshop were chosen to reflect both the computationally intensive kernels and data manipulation requirements of our applications.

Mathematical Model. The solution involves a range of techniques including massive table look-up for integral values and extensive matrix calculations.

Abstract Algorithm and Approach. The algorithms studied are:

A. Sparse Matrix Vector Product: The product of a large randomly sparse symmetric matrix times a set of vectors occurs in many applications. In computational chemistry it occurs in constructing the Fock matrix in Self Consistent Field (SCF) calculations, in solving linear equations, and in solving for the lowest few eigenvalues and eigenvectors.

B. Four Index Transformation: The four index transformation is needed to transform the two electron integral file (a function of four variables, F(i,j,k,l), with respect to a different basis set.

$$G(p,q,r,s)=\sum_{i,j,k,l} C_{p,i}\, C_{q,j}\, C_{r,k}\, C_{l,s}\, F(i,j,k,l)$$

The transformation algorithm involves the formation of partial sums to reduce the computational complexity but it requires a significant shuffling (reordering of the partially transformed integrals) half-way through the calculation to keep the memory references local. To obtain efficient vectorization it is necessary to treat molecular symmetry explicitly, which essentially blocks the function F into relatively dense subunits.

C. Diatomic Slater 2-electron Integrals: The numerical evaluation of $O(n^4)$, where n=100-300, diatomic exponential (Slater) type orbital two-electron integrals. The algorithm uses the Neumann expansion for r_{ij}^{-1} and each term in the expansion involves an iterated double numerical integration. The integrals are required to have a high degree of (absolute) accuracy--typically $<10^{-10}$--to avoid numerical linear dependency problems. A charge distribution approach is implemented. For each term in the expansion the set of n^2 charge distributions(CD) are calculated and all possible vector dot products (length O(500)) are formed. Given sufficient memory to hold all of the charge distribution quantities the CPU time is dominated by the dot products.

The Program. A. The sparse matrix vector algorithm for symmetric matrices is complicated slightly (relative to the nonsymmetric case) by the fact that each element H_{ij} contributes to both D_i and D_j. I/O and memory concerns still strongly suggest that the symmetric form of H be explicitly utilized. One implementation requires each processor to have access to all of C with each processor forming a partial result Dp. When all of the records of H have been processed the product D is calculated as

$$D_{ik}=\sum_p Dp_{ik}$$

The algorithm is easily distributed over the PEs where each PE stores a portion of H in its array memory. The corresponding partial sum Dp is computed by importing the needed elements of the vectors C. Depending upon the size and sparsity of the matrix of the matrix, each PE will need only part of the elements of C. For the n=20000, 1% nonzero test case, each PE will require about 3/4 of the the elements of C. The rate limiting step of this implementation is therefore the network transfer time to transmit C and Dp. For the above example the I/O time would be about 0.16 seconds while the total CPU time would be only 0.01 sec. The total execution time is thus 0.17 seconds.

Another implementation has each PE store about m=n/256 elements of C and of D. The matrix is then divided (sorted) into M*(M+1)/2 subblocks each spanning approximately an equal number of rows and columns (m) with each

subblock allocated to a PE (M=22 square subblock would allocate one block to 253 of the PEs). Each PE would then require no more than 2m elements of C and would calculate no more than 2m elements of Dp. Furthermore, each element of D could be summed requiring no more than M elements of Dp. Thus. the number of words to be transferred over the network for each PE is m(M+2). For our sample problem. m=79 and M=22: the network transfer time is less than 0.01 seconds.

B: The four index transformation. If we define the n_i by n_j matrices F^{kl} as the corresponding subunits of F then the first half transform may be expressed as a sequence of similarity transforms, $H^{kl} = C^T F^{kl} C$. If we shuffle the elements of H to form H' with $H_{kl}{}^{ij} = H_{ij}^{kl}$, then G may be formed as another sequence of similarity transforms. The algorithm is thus broken into the following steps:

1. The expand step to form the matrices. F^{kl}. Only the elements of F greater than some threshold are usually stored on disk. In addition. for many of the symmetry blocks of F there are restrictions on the range of the indices since only the unique integrals are stored.
2. First half transform. denoted as MXM1.
3. Sort or shuffle step to reorder the partially transformed integrals. Since the integral file does not fit into memory, random access is used to perform a bin sort.
4. Second half transform. denoted as MXM2.

The algorithm transfers trivially to a multiprocessor environment because each of the $O(n^2)$ similarity transforms can be performed independently. Assuming the integral file will fit in memory then this algorithm will perform well on the data flow machine. The shuffling required between half transform steps will not dominate the calculation.

C. Two electron diatomic Slater integrals. There are many organizations possible for implementing this algorithm on multiprocessors. Since any number of the $O(n^4)$ integrals may be computed independently, the simplest approach is to partition the integral list and have each processor work on separate partitions. If we define the speedup in performance for n processors to be the ratio of the performance of n processors to that of one processor, then the speedup for this implementation is n since each of the partitions is independent. The implementation could be rather inefficient, however. since most of the CD quantities will need to be recomputed many times. We can divide the $O(n^2)$ CD quantities among the PEs (m per PE) and partition the integral file into subblocks. Each processor would need at most 2m CD quantities, and each would compute $O(m^2)$ integrals. Also, each PE would need to only import the CD quantity itself and not the associated tables needed to form the CD quantity. This implementation is thus expected to perform very well, close to the 1.2 GFLOPS limit, without requiring considerable redundant calculations.

Conclusions. In conclusion, computational chemistry codes can perform well on the static data flow machine [Levi85]. There is a considerable degree of parallelism in the algorithms that can be easily exploited. Considerable algorithmic development will be required for some steps (notably the multiconfiguration self consistent (MCSCF) and configuration interaction (CI) steps) to reduce the network traffic.

The scientists found coding in VAL straightforward but expressed a number of reservations about the programming environment. They also felt that the compiler directive (advice file directives) to effectively map the problem onto the data flow machine might be nearly as hard to write as the VAL program itself.

For the particular machine proposed to the data flow study, disk I/O is significantly underdesigned for chemistry algorithms. The total disk I/O rate is little more than that of a single channel on a Cray and is far less than of the Cray Solid State Disk rate (1.2 GByte per second). The work done for this application assumed that there would be a large buffer memory to synchronize I/O and to allow simultaneous read of sequential records (to different PEs) from a file possibly scattered over many disks.

Some of the chemistry algorithms will not fit in the assumed instruction memory size. The analysis was not carried out to sufficient detail, however, to provide a realistic estimate of the instruction memory space required.

6.4. Galactic Simulation

Team: Eric Barszcz & Cathy Schulbach

Problem Overview. The problem is to simulate the movement of particles, i.e. stars, in a galaxy. Typically, an entire galaxy is simulated. Inputs to the simulation are the initial particle positions and their velocity vectors. There may be more than 100,000 particles. Currently, this code runs on a Cray X-MP/22 and the particle pusher phase takes approximately 0.7 seconds per iteration.

Mathematical Model. A discrete simulation is used. At each time step, the change in particle position is computed based the current position and the surrounding potentials. Ideally, the change is computed at the finest granularity, that is, the effect of every other particle on the current particle.

Abstract Algorithm and Approach. Because of the size of the problem, the ideal solution is not feasible. Instead, break the galaxy into a grid and compute the average potential at each grid point. Use a 64×64×64 grid.

The computation has two phases. First, the potential grid is computed based on the density grid. Second, the particle positions are updated based on both current position and the force on a given particle from the 26 nearest potential grid points. The new position information is then used to compute a new density grid which is used in the next iteration of the computation.

The team rejected transliterating the program from FORTRAN into VAL because the algorithm was not evident in the program. Instead, the team started with the algorithm as specified above, but also questioned this choice in their report, noting that the algorithm may have been chosen because it was easy to implement in FORTRAN. Furthermore, because of the size of the problem, the team chose to implement only the second half of the algorithm, called the *particle pusher*, on the data flow machine. The estimated run-time for the particle pusher on the data flow machine were compared to the 0.7 second one-pass time on a Cray X-MP.

The team chose to store the potential grid data in the local memory of the PEs and then stream the particle data past such that for each particle. The new position computation is performed as soon as its neighborhood is available. The potential grid is too large to fit in the local memory of any one PE, so it must be divided up and placed in more than one PE. The scientists developed an algorithm to determine the effective computation rate based on different dividing schemes.

The Program. The final program developed divided the potential grid into 4×4×4 cubes with boundary layers. Each cube is stored in a separate PE. Then, assuming the particle arrays are presorted, each PE reads only a section of the particle memory. Using this approach, and summing the times to read the particle data, to drain the pipe afterwards, and to bring the results together yields 0.0516 seconds per iteration.

The team then examined the case where the grid size was changed to twice the resolution and the number of particles increased to one million and computed 0.289 seconds per step, or a computational rate of 1.17 GFLOPS.

Conclusions. The particle pusher phase of the galactic simulation adapts well to the MIT static data flow architecture. Array storage is not a serious problem, but since this is only one part of the whole galactic simulation program, the number of PEs and program storage may be too small to hold the entire program.

6.5. Linear Systems

Team: Merrell Patrick

Problem Overview. Sparse linear systems of algebraic equations frequently arise from the numerical approximation of mathematical models used, e.g., in structural analysis, fluid dynamics, and circuit analysis. As the models become more sophisticated and their numerical approximations become more accurate, the linear systems to be solved become very large and quite sparse. Such systems are usually solved using iterative methods [Reed84].

For this study, then, we consider the general problem of solving the linear algebraic system

$$K\ x = f$$

iteratively using the iteration defined by

$$x^{(k+1)} = A\ *\ x^{(k)} + c$$

where $x^{(k+1)}$, $x^{(k)}$, and c are vectors with n components and A is an $n \times n$ matrix whose elements are functions of the elements of K. The matrix A is assumed to be random sparse with r percent of its elements nonzero.

Abstract Algorithm and Approach. The "divide and conquer" paradigm for developing parallel algorithms was used to define two different approaches.

In the first approach the iteration matrix, A, and the constant vector, c, were partitioned into sets $A[i]$ and $c[i]$, respectively, of $n/256$ contiguous rows for $i = 1, \ldots, 256$. $A[i]$, $c[i]$, and a copy of the iteration vector $x^{(k)}$ were assigned to the array memory of PE i. PE i, concurrently with all other PE's, evaluated the expression $A[i] * x^{(k)} + c[i]$ to produce components $(i-1)n/256+1$ to $in/256$ of the new iteration vector $x^{(k+1)}$.

In the second approach, A and c were partitioned and assigned to $PE\ i$ as above. In addition, the set $A[i]$ in the array memory of $PE\ i$ was then partitioned into sets $A[i,j]$ of $n/256$ contiguous columns for $j = 1, \ldots, 256$. In other words, $A[i,j]$ was a block of $n/256$ rows and $n/256$ columns of the original matrix, A. $PE\ i$ then carried about a computation of the form

$$x^{(k+1)} = 0$$
$$\text{for j in [1,256]}$$
$$\quad x^{(k+1)}[i] = x^{(k+1)}[i] + A[i] * x^{(k)}[i]$$
$$\text{endfor}$$
$$x^{(k+1)}[i] = x^{(k+1)}[i] + c[i]$$

However, the algorithm is implemented so that $PE\ i$ finishes its computation after $PE\ i-1$, but before $PE\ i+1$. The motivation for this and other details of the programs for the two approaches is discussed in the next section.

The Program. The program for the first approach was written so that the PEs carried out their computation concurrently. Since each PE owned a copy of the iteration vector, before it could continue with the next iteration it had to send the new components it computed to all other PEs and, in turn, receive new components of the iteration vector from all other PEs. This organization of the algorithm meant a large amount of data had to be communicated amongst PEs between each iteration step and offered the potential for the communication network to become a bottleneck. This potential problem motivated the second approach.

In the program for the second approach, only PE 1 owns a copy of the complete iteration vector $x^{(k)}$. The computation in $PE\ i$ proceeds as follows at a given time step:

if i not equal 1 or 256 then

PE i reads $x^{(k+1)}[j]$ from its array memory
after it has been received from PE i-1
PE i multiplies A[i,j] and $x^{(k)}[j]$ and
adds it to the accumulating sum
PE i forwards $x^{(k)}[j]$ to array memory of PE i+1
if j = 256 then PE i adds c[i] to the accumulated sum
yielding $x^{(k+1)}[j]$ and sends it to the
array memory of PE 1 where it becomes the new
$x^{(k)}[j]$

This shows that at a given time within an iteration, *PE i* only owns $n/256$ components of the iteration vector rather then all n components. Furthermore, the communication of the iteration vector amongst PEs is spread out over the time required for an iteration rather than waiting until the end of an iteration. This implementation also allows for the possibility of overlapping the communication of data amongst PEs with the computation. These two things together reduces the possibility that communication will become a bottleneck. It is also important that synchronization required for checking for convergence of the iteration can be handled more smoothly in the second approach.

Conclusions. The above programs were mapped onto the machine assuming problem parameters of n (number of equations) = 40,000 and r (sparsity) = .04. These assumptions mean there were 64M nonzero elements of A requiring 128M floating point operations per iteration. Since the *A* matrix was assumed to be stored in sparse form 80M words of storage were required. We assumed that the combined memory of the 256 processors were sufficient to satisfy these storage requirements.

A first pass analysis based on floating point operation counts, processor speeds, array memory access times, and interprocessor network capacity indicated that the parallel execution time of the program for the first approach was roughly twice that for the program implementing the second approach. This analysis assumed communication between a PE and its array memory could be overlapped and that the logical functional units in the machine would be able to keep the floating units busy.

A more careful analysis of the data flow graph from which the first program was developed and the number of logical operations required to feed the floating point units showed that performance of the machine was sensitive to the number of logic units. Three to four logic units were needed in order that logical arithmetic not become the bottleneck in the computation. The analysis also clearly indicated that the performance of the machine was affected by the array memory access method used by the machine. In particular, random access of data from the array memories was far superior to sequentially streaming data from the memories through the functional units. Our analysis did not indicate that interprocessor communication would be a bottleneck as we had earlier feared. Details of this analysis can be found in [Reed85a, Reed85b]. A careful analysis

of the data flow graph from which the program for the second approach was written is in progress.

Based on the results we have so far, we believe that with careful design of the functional units and attention to memory access patterns, iterative methods for solving large sparse linear systems of equations can be implemented efficiently on a static data flow machine.

6.6. Artificial Intelligence: Natural Language Processing

Team: Rick Briggs

Problem Overview. A natural language processor is a system that can understand and intelligently respond to natural language input. For this problem, solution techniques, algorithm designs, and performance statistics are under active research and final results are unavailable. For an overview of the topic, see [Bobr75].

Abstract Algorithm and Approach. Natural Language Processing (NLP) can be viewed as a three-part process: understanding, intelligence, and expression. Figure 5 gives the overall structure: Natural language comes in

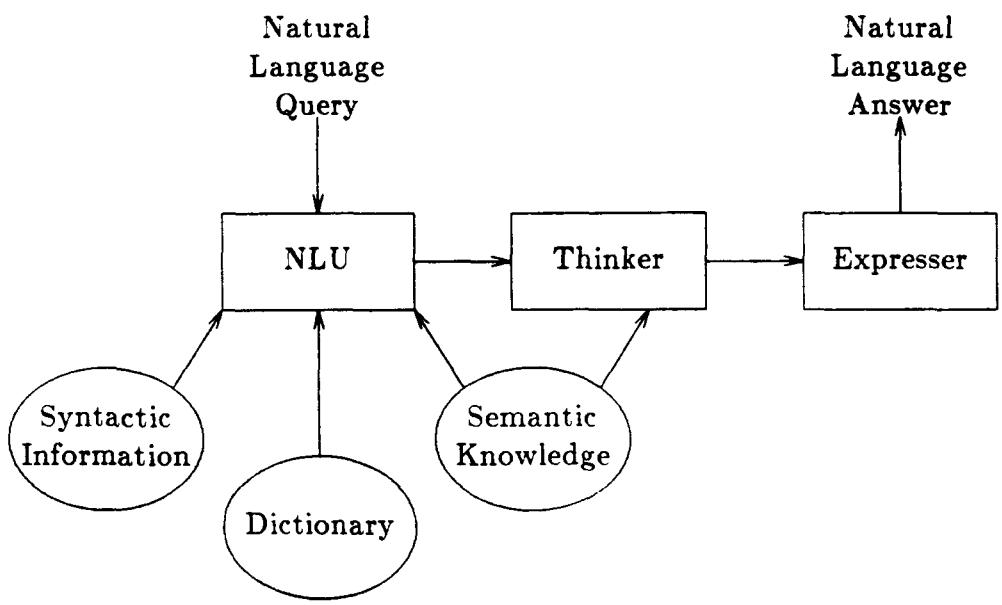


Figure 5. Structure of Natural Language Processor

(either query or text) and the NLU (Natural Language Understander) determines the real meaning of the input, including all semantic nuances not explicitly stated. This is done by consulting:

1. Syntactic information, a relatively small database.
2. Semantic knowledge, the true "knowledge base" of the system in which is stored knowledge of the world to aid in understanding; can be enormous.
3. Dictionary, the English lexicon matched to meanings in representation; there should be at least 100,000 entries with 1 million being more realistic. Each of these entries can contain complicated structures.

For the purposes of this study, only the Natural Language Understanding (NLU) portion was used. A more detailed diagram of its structure is given in Figure 6. Further information on the processing of natural language can be found in [Bobr80, Bruc75, Ries74, Scha74].

A series of searches are necessary to find the appropriate semantic structures and syntactic information of the input. The remaining parts of the computation involve the expansion and filling of "templates," or "cases," where each template has slots to be filled either with primitives, or other semantic tokens that need to further be expanded.

For text analysis there are two possibilities. Either text comes from a human directly or the NLP is hooked up to, say, a news wire and is receiving input over time, as in [Cull78]. In the first case, the processing of different context blocks should be spread out in space, whereas in the last, pipelining is more appropriate.

The Program. A portion of this application was written in Prolog, and was then used for the translation to VAL.

The steps at the top of Figure 6 to find the appropriate semantic structures and extract the syntactic information from the input are first performed sequentially. This step can be performed rapidly using standard sequential methods; it was not encoded in parallel. The main computation of the program, called "generate," takes the form of a **forall** definition of an array, called "case-array" in the program, in which there are embedded **forall** loops for instantiating primitives. The cases are templates with holes that require further instantiation with cases (hence the loop on the "Instantiate Case" box in Figure 6) or have holes to be filled with words in the lexicon.

The trimming step would allow each slot to be processed in parallel by allowing multiple candidates to fill the slots according to only semantic considerations. Later, to select the correct instantiation, syntactic information can be used to "trim." Note that backtracking sequentially, which will be slower, would allow one to fill in the correct slot immediately by examining the other slots, since in this case processing is not done in parallel (although backtracking can be simulated to some extent using a queue).

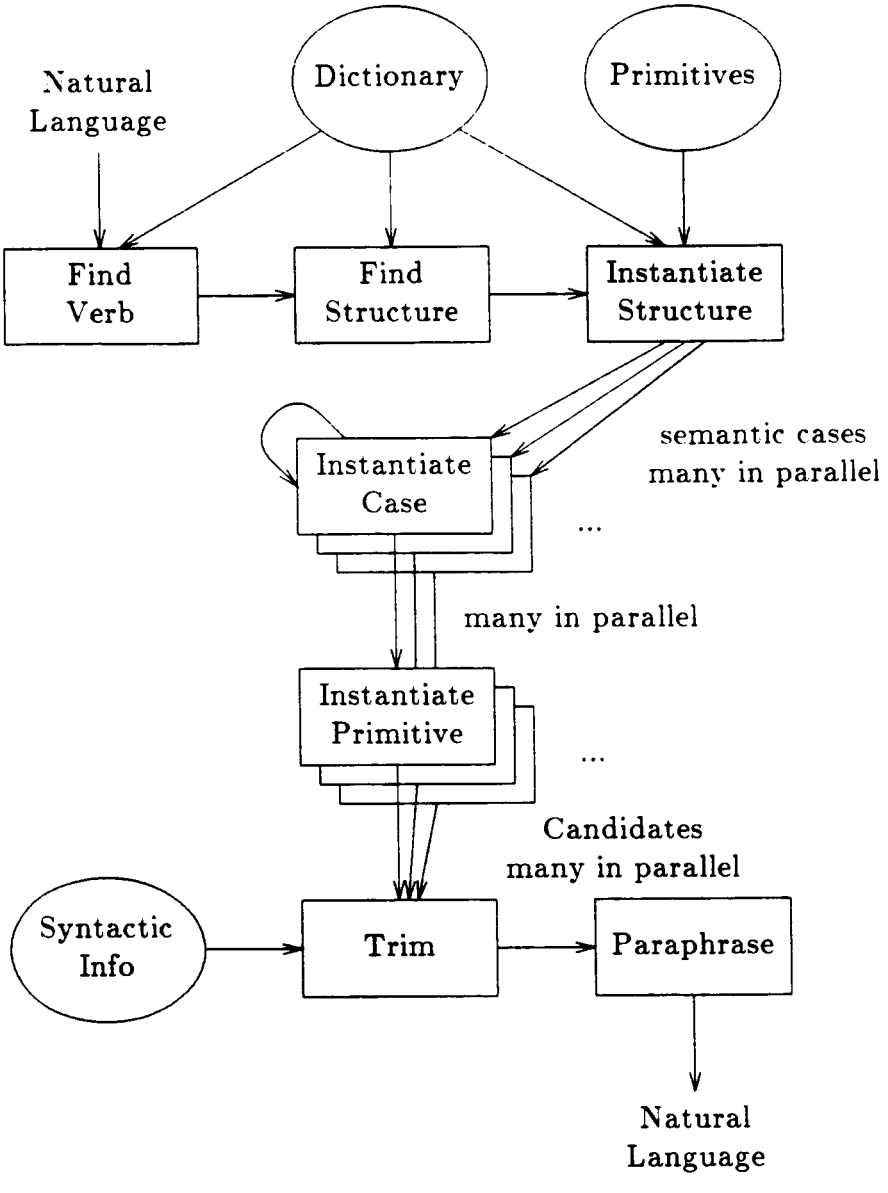


Figure 6. Structure of the AI VAL Program

Finally, these bindings are mapped onto a template and output in a paraphrased (understood) form.

Conclusions.

1. The lack of global databases in the data flow machine was a serious problem.
2. Overcoming sequential thought patterns so familiar in LISP and Prolog is not easy but is ultimately necessary if AI applications of significantly higher performance are to be achieved.
3. The lack of function libraries, debuggers, and other tools (as in InterLISP) is a problem at least in getting AI programmers to accept parallel machines.
4. The inability to perform I/O at the normal level encountered in (say) InterLISP was a problem.

VAL and data flow were less of a problem than the lack of normal libraries, databases, and software tools -- all heavily used in AI applications. The scientist concludes his own report with

> On the positive side, it is a good exercise to think in the way necessary to program in VAL. Almost everybody who has done programming is very used to sequential thinking. And apart from lack of shared memory, the new way of thinking is superior to the old with respect to AI applications for reasons mentioned above.

6.7. Queueing Network Analysis

Team: Ken Sevcik & Peter Denning

Problem Overview. *Queueing Network Models* are tools for deriving performance estimates for computer systems and communications networks [Denn78]. From descriptions of *workload intensities* (the volume of transactions or messages) and *service demands* (the service required on average at each system device, or *center* by a work unit), performance measures such as throughput, average response time, utilization, and average queue lengths can be computed.

Exact solution of a queueing network model can, in general, be obtained from the solution of a system of simultaneous linear equations in which the equilibrium state probabilities are the unknowns. This approach is practical only for networks with very few classes, few devices, and few customers per class. With two classes, five devices, and ten customers per class, the number of equations exceeds one million. For example, using *mean value analysis (MVA)* [Reis80], the throughput and average response time of a single-class queueing network model with N customers and service demands, D_1, D_2, ..., D_K at the K centers is given by:

$$\text{Throughput}: \quad X(N) = N / R(N) \; ; \tag{1}$$

$$\text{Mean Response Time}: \quad R(N) = \sum_{i=1}^{K} R_i(N)$$

$$\text{where} \qquad R_i(N) = D_i \left[1 + \frac{R_i(N-1)}{R(N-1)}(N-1) \right]$$

Fortunately, queueing network models that satisfy restrictions leading to the property of *separability* [Lazo84] can be solved efficiently. Techniques are known for obtaining performance estimates directly, rather than by summing state probabilities over subsets of system states. For one such technique, called *convolution* or *normalization constant analysis (NCA)* [Buze73, Reis75]

$$\text{Throughput}: \; X(N) = G(N-1)/G(N) \; ;$$

$$\text{Mean Response Time}: \; R(N) = N \; \frac{G(N)}{G(N-1)} \tag{2}$$

where $G(N) = g(N,K)$ and $g(n,k) = g(n,k-1) + D_k \times g(n-1.k)$ with $g(0,k) = 1$, k = 0,1,2, ..., K, and $g(n,0) = 0$, n = 1,2, ..., N. This expression for $G(N)$ is efficient for calculating the normalization constant that assures that state probabilities sum to one, and is defined by

$$G(N) = \sum_{\substack{\vec{n} = (n_1, n_2, \cdots, n_K) \\ s.t. \sum_i n_i = N}} \prod_{i=1}^{K} (D_i)^{n_i} \tag{3}$$

For large multiple-class queueing network models, exact solution even by MVA or NCA can become computationally intractable. The number of operations required by both MVA and NCA performed recursively (NCR, using $g(n,k)$) is approximately $4KC\prod_i (N_i + 1)$, where C is the number of classes, and N_i is the number of customers in class i.

Abstract Algorithm and Approach. The team considered three methods of obtaining exact solutions. The first was the MVA recursion (Eqns. (1)), while the others used a normalization constant computed either recursively (NCR) or directly (NCD with Eqn. (3)), and then using Eqn. (2) to obtain the performance measures.

Performance Tables 2 and 3 summarize the results. Space and operations required for algorithm execution on sequential architectures are given. For data flow architectures, estimates of the number of operator nodes in the full data flow diagram are given along with the number of time steps required under three successively more realistic assumptions:

1. an unlimited number of PEs and unlimited fan-in and fan-out at each operator ("ideal" case),
2. an unlimited number of PEs, but operator fan-in and fan-out two or less,
3. only 256 PEs with operator fan-in and fan-out two or less.

For more discussion and justification of the table entries see [Sevc85].

Table 2. Approximate Computational Costs for a Single Class.

	CONVENTIONAL		
	MVA	NCR	NCD
operations	$4NK$	$2NK$	$KL+\log_2 N$
space	K	$\min(N,K)$	K
	DATA FLOW		
	MVA	NCR	NCD
operators	$3NK$	$2NK$	$K(N-L)$
ideal	$5N$	$2(N-K)-2$	2
2-way fan	$N(3-\log_2 K)$	$2(N-K)-2$	$N+\log_2(KL)$
256 PEs	$\frac{3NK}{256}$	$\frac{2NK}{256}$	$\frac{K(N-L)}{256}$
	if $K>256$	if $\min(N,K)>256$	if $L>256$

$$L = \binom{N+K-1}{N}$$

Single Class Case. MVA and NCR are usually thought to have similar computational cost on sequential machines, so the difference between NCR ($O(N+K)$) and MVA ($O(N\log_2 K)$) with limited fan-in and fan-out is interesting. NCR effectively treats all customer population levels in parallel; MVA sums residence times over all centers for population levels separately. For NCD, the products over K devices require $\log_2 K$ time steps, and the summation over all feasible states requires $\log_2 L$ time steps. Because $\log_2 L$ (essentially $\log_2(N!)$) grows much faster than N, NCD is uninteresting for models of realistic size (i.e., $N \geq 4$ and $K \geq 4$). A sum over the massive number of system states requires so many steps when fan-in is restricted that the gain of parallelism elsewhere with NCD is more than offset. Note that NCD is the most highly parallel of the three algorithms, yet given limited fan-in and fan-out it leads to less efficient parallel code, even for quite small models.

In practice, all useful single-class models are such that $4NK < 10^7$, and they can be solved interactively on conventional machines.

Multiple Class Models. With fan-in of two, NCR is again better than MVA (assuming $\sum_c N_c$ exceeds K) because it avoids recursion over customer population levels. NCD remains acceptable (even preferable!) for slightly larger

Table 3. Approximate Computational Cost for Multiple Classes.

	CONVENTIONAL		
	MVA	NCR	NCD
operations	$4CK\prod_c(N_c+1)$	$CK\prod_c(N_c+1)$	$KL+K\sum_c \log_2 N_c$
space	$K\prod_c(N_c+1)$	$K\prod_c(N_c+1)$	$\sum_c KN_c$
	DATA FLOW		
	MVA	NCR	NCD
operators	$6CK\prod_c(N_c+1)$	$CK\prod_c(N_c+1)$	$K\left[LC+\sum_c N_c\right]$
ideal	$6\sum_c N_c$	$2K$	2
2-way fan	$2\log_2 C-\log_2 K\sum_c N_c$	$K[\log_2 C+\sum_c \log_2(N_c+1)]$	$\max_c N_c+\log_2(CKL)$
256 PEs	$\dfrac{6CK\prod_c(N_c+1)}{256}$ if $\sum_c N_c>8$	$\dfrac{CK\prod_c(N_c+1)}{256}$ if $\dfrac{CK\prod_c(N_c+1)}{N_{max}}>256$	$\dfrac{K\left[LC+\sum_c N_c\right]}{256}$ if $L>256$

$$L=\prod_{i=1}^{C}\binom{N_i+K-1}{N_i}$$

models than in the single class case, because the more classes there are, the longer L stays small relative to $\prod_c(N_c+1)$.

All models so large that they cannot easily be solved interactively on sequential architectures have enough parallelism that their solution on a data flow architecture is constrained only by the number and speed of the PEs (assuming a compiler effective at balancing PE workload). The total number of operations required depends on the number of devices, classes, and customers per class. The number of devices is a multiplicative factor, so with a 1000-fold computational speedup, models with 1000 times as many centers can be solved. The number of classes and customers per class affect required operations exponentially. Consequently, only much smaller changes in the number of classes (unless class populations are very small) or in the number of customers per class (unless

the number of classes is very small) can be supported by 1000-fold speedup. Thus, practically, it is not clear that data flow architectures will significantly assist solution of queueing network models.

Conclusions.

1. Practical space and parameterization limits restrict the size of useful single-class models to those that can be solved interactively by MVA or NCR on sequential machines.
2. MVA or NCR algorithms on data flow machines would make it possible to solve multiple-class models with a few more classes, or with somewhat larger class populations.
3. The most parallel abstract algorithm (NCD) did not lead to efficient parallel code due to fan-in/fan-out restrictions.
4. While models can be larger (more classes, larger populations), the difficulty of generating parameters means that useful models will not grow significantly.
5. This work focussed on "exact" solutions of queueing network models. If close approximate solutions are acceptable (they usually are: model parameters are seldom known precisely), then use of approximate algorithms [Schw79, Chan82], which have computational costs independent of class populations (such as $O(KC)$ or $O(KC^3)$), is appropriate. These algorithms execute acceptably on today's sequential computers.

7. What We Learned: Comments from the MIT Group

The RIACS Data Flow Evaluation Study has been an important contribution to the evolution of the MIT Static Architecture into a machine design that can be successfully applied. The study confirmed that data flow machines can achieve high performance in important practical problems, and that the programming methods and concepts necessary to make effective use of them are readily accessible to scientists experienced in the application of conventional high performance machines. The study augments and reinforces program analysis studies that have been conducted on five applications at MIT [Denn84a], but without the guidance of experts in the application fields.

Users of the study report should understand that the machine configuration specified for the study was chosen only to provide a specific target for evaluation. Indeed, one advantage of data flow architecture is that the number of processors and memory units may be chosen to fit users' needs. We imagine that machines with 16, 64, and 256 PEs might be found well matched to various problem domains. For example, the multi-grid CFD computation appears well matched to a data flow machine with 16 PEs. Furthermore, the size and transfer rates for the array memory and disk storage can be adjusted without making fundamental changes to the architecture. In fact the objective of chosing a specific configuration for the study was to determine which parameters of the design were limiting applicability of the machine.

We were gratified that the teams found our programming language VAL easy to learn, and that, by the beginning of the second week, they were in good command of the issues involved in structuring machine code for high performance on the data flow machine.

Several comments made by the teams merit a response; these deal with the specified machine configuration, provisions for program debugging, the applicability of array memory to global data bases and matrix transposition, and the applicability of VAL and static data flow computation to artificial intelligence problems.

7.1. The Machine Configuration

From several of the teams we learned that the capacity and transfer rates specified for the array memory and disk systems of the machine were limiting the performance achievable for their applications. Some problems can benefit significantly from large amounts of array memory at each PE. Since the time of the study we have found it desirable and practical to provide for much larger array memory capacity (several million words or more for each PE), and a transfer capability of 5 million (64-bit) words per second for each PE. We also learned that disk storage transfers through the routing network would be a bottleneck in applications involving very large amounts of data. In response, we are now planning that communication with disks will be supported by direct connections between PEs and disk control units, each connection supporting 5

million words per second or more for one PE.

7.2. Program Debugging

Considerable concern was expressed by team members about the debugging of VAL programs, specifically the difficulty of requesting output of internal variables, arguments, and results of functions invoked during program execution. In the lectures at RIACS, we did not cover the program tracing facilities of VALSys, and are not sure to what extent the teams made use of these facilities. An important facility not provided by VALSys is the ability to "break" execution of of a data flow program at an arbitrary point (a variable definition, say). This is a feature that could be added to the VAL interpreter, and one that will be supported for compiled programs by a simple hardware feature that will permit insertion of "breakpoints" into data flow machine code. The PE will be designed so any instruction it holds may be altered so that, after its execution, it sends an information packet to the host computer instead of signaling its successor instructions. Following analysis of the computation state by a debugger program (all information on which the breakpoint instruction depended will be static), the debugger can cause the host to send a command to the PE that causes the breakpoint instruction to signal its successors. Of course implementation of the debugger requires information from the compiler about the relationship of variable names in the VAL program to the locations in the machine of the data values and the data flow instructions that produce them. The ability to provide this information will be an important feature of the compiler.

7.3. Array Memory

The Artificial Intelligence team criticizes the absence of "global memory" as making access of a global data base of stored knowledge difficult. The Computational Chemistry team criticizes the absence of "shared memory" in view of the importance of support for transposing multi-dimensional matrices and argues that special hardware should be provided for this purpose. Both of these remarks concern problems that affect any highly parallel machine. One is the problem of distributing a large data base over many processing nodes. Either all requests to access the data base must pass through a single "monitor" or "guardian" of the data base, or the data base must be divided up in some way that permits accesses to different sections to be handled by several monitors. The techniques that have been proposed to implement either strategy are as usable in data flow computers as in other forms of distributed computer organization. We believe proper attention to hardware support can make data flow machines more attractive than other parallel architectures for such applications.

The transposition of matrices is a classical hard problem for parallel computers. Its efficient support does not depend so much on "shared memory" specifically as it does on efficient communication among processors. The ability of the proposed data flow machine to do matrix transposition could be improved by increasing the capacity of the routing network. This is planned in any case,

since we have found that the amount of hardware in the specified routing network is small in comparison with the hardware used to build the PEs. However, a large increase in performance for matrix transposition does not appear warranted because in most cases where transposition is used some computation can be overlapped with data movement operations.

7.4. Applicability to AI Problems

Skepticism over the applicability of VAL and the static data flow architecture to artificial intelligence applications is understandable since our efforts on data flow computation have been directed specifically at supporting "scientific" computation. Actually, many workers in the field have argued that artificial intelligence problems offer a high degree of parallelism. Yet there is no generally accepted programming language or methodology for expressing AI problems for massively parallel computation. The mainstay AI programming language Lisp and the programming style currently used with Lisp machines will not be effective on highly parallel computers. Neither will the logic language Prolog as presently cast.

The principal comment from the AI team is that VAL is not sufficiently "high level" to be a suitable AI language. We are not certain what the scientist means by "high level," but one possibility is that he is disappointed by the absence in VAL of facilities to support data abstraction. The addition of such facilities would make the programming of AI problems "more natural" and this aspect will be considered for future revisions of VAL. Yet this change in VAL would not increase the "expressive power" of the language.

It may be that the AI scientist is concerned that too much writing is required to express an algorithm in VAL as compared to Lisp -- primarily due to the strongly typed nature of VAL. This will be less of a problem once automatic type inference is incorporated in the VAL compiler, as this will relieve the programmer from having to write most type declarations, while maintaining the benefits of type checking by the compiler.

We were pleased to witness the experience of expressing a reasonably authentic artificial intelligence problem in the VAL language and to become acquainted with the issues of exploiting the power of a data flow computer in its execution. We believe that many AI problems will be solvable with high performance using the static data flow architecture.

8. Conclusions

The following are the conclusions from the study about VAL and the proposed static data flow computer architecture.

8.1. Programming

8.1.1. New Approaches

One of the questions addressed by the study was, "To what extent would data flow concepts suggest new approaches to problems?" To achieve maximally parallel algorithms, Jack Dennis advised participants to return to basic principles: identify the mathematical model underlying the solution, express the data dependency graph of the solution as part of an abstract algorithm, and finally map the abstract algorithm into the data flow language. In terms of Figure 1, he advised participants to back up to Stage 1 of the problem-solving process rather than transliterate a Stage 2 algorithm or Stage 3 program.

We observed that in six of the seven problems, the mathematical model is well understood and the abstract algorithm is already an expression of a highly parallel solution. Hence six of the teams spent their efforts on mapping their abstract algorithms into VAL, and the VAL algorithm into program-graph notation for compilation. Only in the AI problem was the mathematical model sufficiently undeveloped that a retreat to this stage of the problem solving process may be worthwhile.

It is reasonable to suppose that for well understood problems the mathematics have evolved to the point that the abstract algorithm is already in a highly parallel form. This finding may not generalize outside the study, however, because there are other problem domains where highly parallel abstract algorithms have yet to be fully developed. Moreover, one team (Queueing Networks) showed that the most parallel abstract algorithm does not lead to the fastest data flow algorithm.

Another implication of this finding is that future evaluations of this kind may want to pay special attention to being sure that the participants have indeed demonstrated beyond reasonable doubt that their abstract algorithms are in fact highly parallel.

8.1.2. Programming Environment

The MIT translator and interpreter were developed in 1979 and implement what is no longer modern programming technology. One shortcoming is the lack of assistance for writing, managing, or debugging VAL programs. This was seen as a serious shortcoming by several of the participants. One scientist pointed out that even the FORTRAN-style of debugging programs, using WRITE statements to display intermediate results, is cumbersome in VAL.

VAL is designed to express algorithms that use a pure data flow model of computation (see Section 5.1). However, the static data flow architecture

considered in the study does not implement this ideal model. In particular, the machine includes array memories, to support high performance operation, that are not part of the model. The array memories are outside the scope of the model and also outside the scope of VAL: a programmer cannot control array memory activity via a VAL program. Several study team members were uncomfortable with this fact because they felt that they would need to control array memory usage to achieve good performance of their application code (with respect to the best performance that could be achieved). An "advice file" companion to a VAL program could be used to provide a VAL compiler with information to guide its allocation of array memory.

Although program graphs play a strong role in the abstract algorithm and again in the mapping of VAL to machine code, there is no interactive graphics in the programming environment. Several team members indicated that it might be convenient if they could just draw the data flow graphs and have a graphical editor and compiler translate them into the language of the machine. We envision that a programming environment could be constructed based on direct rendition of data flow programs as graphs. The new language ("Visi-VAL"?) might complement the current flat language, VAL, and is worth investigating.

8.1.3. Learning Effort

Although they had no prior experience with data flow machines, the programmers participating in the study took only a very short time to learn VAL well enough to use it in their disciplines. (Most had learned VAL within two days rather than five days as we had anticipated.)

In the AI area, there appears to be a much greater reliance on function libraries (e.g., in LISP) and on software tools (e.g., run-time debugging aids and powerful graphics interfaces) than in the scientific computing areas. The AI team's criticism of the data flow machine is based on the lack of libraries and tools rather than on skepticism toward parallel programs. We conclude that the data flow concepts are sufficiently intuitive and well represented by VAL that they are easily grasped. It is likely that this was aided by the fact that the majority of abstract algorithms were already in a highly parallel form.

8.2. Architecture

8.2.1. Array Memory Bandwidth

The Array Memory is the most problematic part of the architecture. Arrays arise naturally because most of the abstract algorithms pass matrices from one computational phase to another. The participants were initially told to ignore the representation of arrays and instead think of the abstraction that an array is stored inside a data token. Because in their experience the efficiency of their algorithms depends critically on how array data is represented, the participants did not generally accept this abstraction.

The bandwidth of the Array Memory is a concern. The aggregate bandwidth available to the 256 PE machine is 1 GByte per second. This is comparable to that available from a Cray X-MP(4) Solid State Disk (2.5 GBytes/sec.). However, the individual bandwidth available to a PE is only 2 Mwords per second. Frequent access to the array memory by a PE (clock rate of 200ns) may significantly influence the achieved processing speed of the PE. Some of the algorithms investigated in the study used enormous amounts of array memory, some in the form of very large sparse matrices. The MIT researchers have estimated that one of twenty memory references accessed an array element. In practice, we found this ratio to be much higher.

8.2.2. Array Memory Function

Among the most important operations on arrays is transposition: the computational phase that consumes an array may require the elements to flow in different orders on program-graph edges from the orders in which they were generated. Transposition is handled in conventional machines by random-access memory (RAM) -- the producing phase stores the array elements in memory shared with the consuming phase. In a data flow machine, however, there is no shared memory.

We conclude that the architecture needs to explicitly incorporate hardware for transposing array data. This can be done with special purpose sorting networks or with a staging memory such as that in the MPP. A clearer specification of array memory and transposition operations needs to be developed.

8.2.3. Input/Output

The nature of the load on the interconnection network providing communication between the PEs and between the data flow machine and its disks is important. Knowledge of the characteristics of the load are essential to the process of choosing an effective network. In particular, the capability to support array transposition and high-bandwidth disk communication was found to be essential. As a point of comparison to support these statements, the disks typically used on a Cray X-MP are capable of transferring data at 80 Mbits/second and with ten of them connected to a single I/O processor and operating simultaneously, the system is capable of 500 Mbits or more per second. While the aggregate bandwidth of the routing network is adequate to support a comparable data rate, this requires all 256 I/O ports of the network versus one port on a Cray. Thus to achieve comparable I/O bandwidth on the static data flow machine would require data set partitioning over many more disks than the Cray. For this reason some study participants felt that high I/O bandwidth was not as readily available as on the Cray.

8.2.4. Processing Element Design

The Linear Systems team expended some effort assessing the quantitative requirements for logical operations and floating point operations in their application. They found that overall machine performance was sensitive to the ratio of logic to floating point function units in each PE. For the linear systems application a ratio of three to four logic units to one floating point unit is needed for best performance. Essentially, a match of the number of various function unit types in each PE with the relative numbers of operation types called for by a data flow graph is needed so that one type of function unit is not a bottleneck. This issue deserves further attention.

8.2.5. Cost

We estimate that the commercial cost of the machine would be in the range of $5-10 million. This allows for labor, software, and marketing costs that would be folded in. No software costs are included in these estimates. (See Appendices.)

We recommend that a more careful study be made of the costs of building a data flow machine and of procuring one commercially.

8.3. General

8.3.1. Scaled-Down Machines

Two of the seven problems were unable to take full advantage of the full power of the machine (1.28 GFLOPS). This appeared to result from the mathematics of the problems themselves rather than from failure to begin with a highly parallel abstract algorithm. On the other hand, five of the seven problems consumed all the computing power that was available.

We conclude that scaled-down versions of the machine (e.g., with 16 PEs) should be constructed so that domains requiring less power can obtain it at a fraction of the cost of the full machine. For example, a 16-PE machine would be comparable to a DEC VAX 11/780 in cost but would be capable of solving problems in certain domains that now must use a Cray-class computer.

8.3.2. Cost Performance Advantage

It is a well-known rule of thumb for many of the study team members that a Cray computer can readily deliver one-fourth of its rated peak speed to most applications; greater speed is often obtainable at the price of increased programming effort, including coding program sections in assembly language. Such rules of thumb do not yet exist for static data flow computers.

For the problems studied, the cost-performance advantage of the data flow machine appears significant. We estimated speed-up factors of roughly 10:1 compared to today's Cray computers. This ratio compares the best speed estimated for the data flow machine (1.28 GFLOPS) to the best speeds seen for

applications codes on Cray-class computers (roughly 100 MFLOPS). Given a data flow machine price estimate of as low as $5 million, it may have as much as a 2:1 price advantage as compared to a Cray-class computer. Thus, we can estimate an overall gain of up to roughly 20:1 in cost per solution with the static data flow machine.

8.4. Further Work

We see two areas where further work is required: deeper analysis of the MIT static data flow machine and investigation into developing techniques for evaluating new concurrent architectures. Each of these can proceed independently, with knowledge gained in one contributing to the other.

In the first category, our workshop leaves gaps in the evaluation-picture of the machine we studied. Further work requires the development of a prototype, or better, an emulator for the static data flow machine so that complete applications can be compiled and executed. This would allow direct cost and performance comparisons to existing supercomputers. An emulator would also allow machine "tuning," that is, investigation of the effects of changing machine parameters such as array memory size, network structure and speed, and processing element structure.

The second category, developing and applying general methodologies for evaluating new architectures, could provide a sound basis for comparing concurrent machines as they are proposed an developed. Our study of the static data flow machine convinced us that taking real-world problems and implementing them on the new machine is the best approach and yields the most valuable results. We feel that much work could be done here and that our study is a good first step.

9. References

[Acke79]
Ackerman, W. B. and J. B. Dennis, "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual," MIT/LCS/TR-218, Massachusetts Institute for Technology, Cambridge, MA (Jun 1979).

[Acke82]
Ackerman, W. B., "Data Flow Languages," *Computer* **15**(2), pp. 15-25 (Feb 1982).

[Acke84]
Ackerman, W. B., "Efficient Implementation of Applicative Languages," LCS/TR 323, MIT (Mar 1984).

[Adam84]
Adams, G. B. and P. J. Denning, *A Ten Year Plan for Concurrent Processing Research*, RIACS (Mar 1984).

[Adam85]
Adams, G. B., R. L. Brown, and P. J. Denning, "On Evaluating Concurrent Architectures," RIACS Technical Report, RIACS (May 1985).

[Bobr75]
Bobrow, D. G., A. Collins, and editors, *Representation and Understanding: Studies of Cognitive Science*, Academic Press, New York (1975).

[Bobr80]
Bobrow, R. J. and B. L. Webber, "Knowledge Representation for Syntactic/Semantic Processing," *1st AAAI*, (1980).

[Bruc75]
Bruce, B., "Case Systems for Natural Languauge," *Artificial Intelligence* **6**(4), pp. 327-360 (1975).

[Buze73]
Buzen, J. P., "Computational Algorithms For Closed Queueing Networks With Exponential Servers," *CACM* **16**, pp. 527-531. (Sep 1973).

[Chan82]
Chandy, K. M. and D. Neuse, "Linearizer: A Heuristic Algorithm For Queueing Network Models Of Computing Systems," *CACM* **25**, pp. 126-133. (Feb 1982).

[Cull78]
Cullingford, R.E., "Script Application: Computer Understanding of Newspaper Stories," Computer Science Research Report #116, Yale (1978).

[Davi82]
Davis, A. L. and R. M. Keller, "Data Flow Program Graphs," *Computer* **15**(2), pp. 26-41 (Feb 1982).

[Denn78]
Denning, P. J. and J. P. Buzen, "The Operational Analysis of Queueing Network Models," *Computing Surveys* **10**, pp. 225-261. (Sep 1978).

[Denn83]
Dennis, J. B. and G. R. Gao, "Maximum Pipelining of Array Operations on Static Data Flow Machine," *Proc. 1983 Int'l Conf. on Parallel Processing*, (Aug 1983).

[Denn84a]
Dennis, J. B., "Data Flow Ideas for Supercomputers," *Proc. COMPCON '84 Conf.*, , pp. 15-19 (Feb 1984).

[Denn84b]
Dennis, J. B., G. R. Gao, and K. W. Todd, "Modeling the Weather with a Data Flow Computer," *IEEE Transactions on Computers* **C-33**(7)(July 1984).

[Dias81]
Dias, D. M. and J. R. Jump, "Analysis and simulation of buffered delta networks," *IEEE Trans. Computers* **C-30**, pp. 273-282 (Apr 1981).

[Eber85]
Eberhardt, D. S. and K. Rowley, "An Analysis of Static Data Flow to a Sample CFD Algorithm," Technical Report, RIACS (Mar 1985).

[Gao82]
Gao, G. R., "An Implementation Scheme for Array Operations in Static Data Flow Computers," MIT/LCS/TR-280, Massachusetts Institute for Technology, Cambridge, MA (May 1982).

[Gurd85]
Gurd, J. R., C. C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *CACM* **28**(1), pp. 34-52 (Jan 1985).

[Lazo84]
Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik,, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Englewood Cliffs (1984).

[Levi85]
Levin, E., "Performance Evaluation of a Static Data Flow Processor for Transformations of Large Arrays," TR 85.1, RIACS (Jan 1985).

[Merr85]
Merriam, M. L., "Application of Data Flow Concepts to a Multigrid Solver for the Euler Equations," *Proc. 2nd Copper Mountain Conf. on Multigrid Methods*, (Apr 1985).

[Reed84]
Reed, D. A. and M. L. Patrick, "A Model of Asynchronous Algorithms for Solving Large, Sparse, Linear Systems," *Proc. 1984 Int'l Conf. on Parallel*

Processing, (Aug 1984).

[Reed85a]
Reed, D. A. and M. L. Patrick, "Iterative Solution of Large, Sparse Linear Systems on a Static Data Flow Architecture: Performance Studies," RIACS Technical Report, RIACS (Feb 1985).

[Reed85b]
Reed, D. A. and M. L. Patrick, "Model of a Data Flow Architecture for the Iterative Solution of Large, Sparse Linear Systems," *Proc. 1985 Int'l Conf. on Parallel Processing*, (Aug 1985).

[Reis75]
Reiser, M. and H. Kobayashi, "Queueing Networks With Multiple Closed Chains: Theory and Computational Algorithms," *IBM J. R&D* **19**, pp. 283-294. (May 1975).

[Reis80]
Reiser, M. and S. S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," *JACM* **27**, pp. 313-322 (Apr 1980).

[Ries74]
Riesbeck, C. K., "Computational understanding: Analysis of sentences and context," AI Laboratory Memo 409, Stanford (1974).

[Rumb77]
Rumbaugh, J., "A Data Flow Multiprocessor," *IEEE Trans. Computers* **C-26**, pp. 138-146 (Feb 1977).

[Scha74]
Schank, R. C. and C. Rieger, "Inference, and the Computer Understanding of Natural Language," *Artificial Intelligence* **5**, pp. 373-412 (1974).

[Schw79]
Schweitzer, P. J., "Approximate Analysis of Multiclass Closed Networks of Queues," *Proc. Int'l Conf. on Stochastic Control and Optimization*, (1979).

[Sevc85]
Sevcik, K. C. and P. J. Denning, "Execution of Queueing Network Analysis Algorithms on a Data Flow Computer Architecture," RIACS Technical Report, RIACS (1985).

APPENDICES

10. Machine Cost Estimate

The hardware cost of the MIT data flow machine as specified for the study will likely be in the rough neighborhood of the cost of one of today's commercially available supercomputers. This estimate is based on the following considerations.

First, the majority component of the machine is memory. The array memory consists of 64 Mwords of storage: this is eight times larger than typical main memory sizes of current supercomputers. Main memory dominates the cost of current supercomputers for machines with medium to large memory configurations. Even though the Array Memory uses relatively slow memory, its cost will dominate the hardware cost of the data flow machine. For the data flow machine to be as cheap as current supercomputers, its memory must be one eighth the cost per word. This cost is possible: significantly less is not.

Second. while the MIT data flow machine uses inexpensive PEs, it uses many of them. It is entirely reasonable to assume that 256 data flow machine PEs will not cost significantly less than the CPU cost of current supercomputers. Thus, there is little overall cost advantage with respect to CPU.

Finally, peripherals for the data flow machine and a current supercomputer will embody the same technology. Also, there is no reason to suppose that a data flow machine will require fewer peripherals than existing machines, given a comparable mission. Hence, peripheral costs are likely to be similar.

Thus, one can forecast that the MIT data flow machine specified in this study, made available commercially, to cost on the order of $5 - 10 million. Total costs including site preparation, software development, and operations could easily be $20 - 30 million over a three to four year lifetime. Site preparation should be simpler that for current supercomputers due to the reduced cooling requirements of the technology proposed for constructing the data flow machine.

11. Questionnaire and Responses

Participants in the data flow study were presented with a questionnaire to assess their subjective impressions and opinions. The questionnaire is presented one question at a time and the individual attributed responses follow.

Question 1: What background of knowledge about data flow and VAL did you have before the Study?

Briggs (AI):

> No background in VAL, basic theoretical acquaintance with data flow in general.

Eberhardt (CFD):

> My previous experience with data flow was a naive understanding of the basic principles (activity templates, etc.). My background in applicative languages, such as VAL, was non-existent.

Levin (Chemistry):

> None.

Merriam (CFD):

> I had seen several lectures, talked to Jack Dennis personally on one occasion. Also I read all the papers provided before the class.

Partridge (Chem.):

> I had read some of the data flow literature--would describe my background as having a basic understanding of what data flow is and what some of the major concerns are.

Patrick (Lin. Sys.):

> I had read three to four papers on data flow languages and architectures.

Question 2: Please list the programming languages and computers with which you would be comfortable attempting a programming task as part of your regular job activities. In a separate list name those languages and computers you most often use on the job.

Briggs (AI):

> Programming languages with which I am comfortable: LISP, Prolog, C, Pascal, Basic, DB Query Languages. Languages most often used: Prolog, C.

Eberhardt (CFD):

> FORTRAN is the language that I have most experience in, but occasionally I use Pascal. Pascal has too many problems when dealing with large numerical array structures so generally I avoid it. Most of the NASA work I have done has been on the VAX. I have not used the Cray X-MP extensively but I understand it well. I anticipate that my future activities will draw heavily on the Cray resource. If an applicative language were available, with a good and trustworthy compiler I think I would use it instead of FORTRAN. First, however, I/O will have to be changed so that information can be extracted directly from a nested function. In response to the question of what machine I would feel comfortable with, my answer is that I am trying to understand as many architectures as I can since it is my job to try to

develop algorithms for them.

Levin (Chemistry):

(a) FORTRAN. (b) FORTRAN; Cray X/MP, CDC Cyber 205, VAX 11/7XX.

Merriam (CFD):

Languages: FORTRAN*, Vectoral*, Pascal, CFD, Lisp, Basic, Compass (assembly language), MRS. Computers: Cray X/MP-22*, CDC 7600, Cyber 205, Illiac IV, VAX 11/7xx*, IBM 360/67, CDC 6400, IBM 1800, DEC-20.

Partridge (Chem.):

Languages: (with an estimated competency level--10 know fluently) FORTRAN (10), Pascal (5), assemblers (Cray, VAX, CDC 7600) (4). At one time I programmed in Basic, PL/1, and Algol. Computers: Cray XMP, CDC Cyber 205, VAX-11/780 (VMS), CDC 7600 (SCOPE), CDC 835 (NOS). In work related applications almost everything has been in FORTRAN. Assemblers only needed for coding small kernels and in debugging.

Patrick (Lin. Sys.):

Most of my programming experience is with FORTRAN, PL/I, and Pascal running on IBM mainframes and PCs. I have some experience using the early FORTRAN which ran on the Cray 1. More recently I have been involved in the development of a parallel programming environment known as PISCES (Parallel Implementation of Scientific Computing EnvironmentS). My primary role has been to implement some classical numerical linear algebra algorithms using PISCES and to give feedback to the primary designer of the system (T.W. Pratt) concerning its usability in scientific computations. PISCES is currently being implemented as an extension to FORTRAN 77. A PISCES program is translated by an interpreter into executable FORTRAN 77 code. The interpreter is currently running under UNIX on a VAX 750.

Question 3: What do you like and dislike about the language constructs in VAL? Why?

Briggs (AI):

The only aspect of VAL I can say I "like" is that algorithms which exploit parallelism are encouraged. I found programming in a low-level language like VAL to be extremely awkward, especially in high-level symbol manipulation. My need to encode a series of sequential computations resulted in a series of definitions in a "let" block which did the work, followed by a trivial "in" component, which simply returns the answer. My impression is that VAL is designed for FORTRAN programmers, and the designers did not really have AI applications in mind.

Eberhardt (CFD):

At this point feel I can rave about VAL. I felt that programming could be handled in a logical and precise manner which reflects the notation of the mathematical algorithm. I can understand and appreciate why Bill Ackerman believes he can work his miracles with his compiler because the language virtually spits out all of the concurrency in the code. My experience with developing concurrent codes in FORTRAN really drives this point home. On the negative side, as a FORTRAN programmer I do not want to relearn syntax. I have never liked the semicolon at the end of each record that free style languages require. Nor do I like having my main, outer shell of my program at the bottom instead of the top where it belongs. Those are minor problems that I can get used to. There is, however, one

serious problem that must be corrected for the language to be at all useful for debugging: the I/O problem. If you want to look at a specific variable in a nested function within a **forall** loop, forget it. You have to pass the argument out through the calling function, requiring global program changes. Also, as pointed out by Harry Partridge, a **forall** loop would have to be converted to a **foriter** loop if only one instance of the array calculation were needed. This must be fixed!!!!!

Levin (Chemistry):

(1) Seem to work. (2) Easy to learn.

Merriam (CFD):

Many of the features of VAL that are nice are also features of Vectoral. These include: dynamic memory, unlimited identifier name length, structured constructs, parallel constructs, strong typing, also (absent from Vectoral) no side effects. It lacks: recursion, synonyms (not equivalence), ability to make a name synonymous with a number in a certain context, ability to iterate over an enumerated type.

Partridge (Chem.):

My major concerns with VAL are:

(a) not convinced that I have to completely (or even largely) give up compatibility with FORTRAN to effectively utilize a data flow machine. The codes are large and it would take an enormous effort to translate them to a VAL-like language. While admittedly I am comfortable with FORTRAN and rather inexperienced with VAL, I do not see that VAL offers me any significant advantages in describing, coding, or debugging my applications. VAL's major advantage is that it is apparently easier to write an optimized compiler.

(b) It appears that even minor changes in a code can involve rather major coding changes, i.e., codes in VAL are not easy to modify. Examples are: adding counters would change **foralls** to **for iters** and debug modifications might involve significant changes in calling sequences.

(c) Memory management capabilities are not conveniently present. These are important if the memory requirements exceed the machine size of localization of memory references needs to be (artificially) enforced.

(d) I/O capabilities are non-existent. No print, format, namelist, buffer in, buffer out, direct access files, et cetera.

Other comments (not in any order).

(a) Debugging looks like it would be difficult. Need a sequential mode.

(b) I liked the **forall** construct but found the **for iter** construct very painful to use. This is particularly true for nested **for iters**. Labels would help make things easier to read.

(c) Surprisingly, I did not find the single definition rule much of a handicap. I'm not sure this is consistent with the comment on the **for iter** construct but . . .

(d) The external statement is a royal pain!

(e) Syntax. (1) Having to end (some) lines with a ";" is a nuisance. Most lines do not continue. (2) The symbol used most frequently is the := for define or replacement, yet it requires three keystrokes to type. I much prefer the = sign (only one keystroke). (3) I don't particularly like the ~ and | symbols for not and or.

(f) I would like a **common** statement at least for parameters that will not be changed in subsequent function calls.

(g) VALSYS aborts with the first syntax error and does not produce a useful listing (formatted).

(h) VALSYS kicks you out for certain errors and array input is very cumbersome.

Why strong typing on input?

Patrick (Lin. Sys.):

The **forall** construct is excellent for expressing natural concurrency within a computation and any parallel programming language should have such a construct. I found the **for-iter** construct awkward to use. Clearly such a construct is needed, given the iterative nature of many scientific computations. However, the advantages of the **for-iter** over **repeat-until** or **do-while** is not clear given the experience of this study.

Question 4: In what ways is VAL better for your application than the language(s) you typically use? In what ways is VAL worse?

Briggs (AI):

I cannot really say VAL has any advantages over LISP and Prolog for AI programming. I cannot imagine writing a significant AI program in VAL.

Eberhardt (CFD):

See answer to Question 3

Levin (Chemistry):

No apparent advantage for my problem over FORTRAN.

Merriam (CFD):

VAL is clearly better than FORTRAN but then, so is almost anything. VAL has no advantage over Vectoral that I can see. On the minus side VAL: doesn't exist, has no I/O and no disk, suffers from inability to pass arguments several levels deep without having them in all intermediate routines, and doesn't allow recursion.

Partridge (Chem.):

VAL, or any functional language, is better only if needed to obtain supercomputer preformance on a machine.

Patrick (Lin. Sys.):

(a) A really nice feature is not having to dimension arrays when writing the program and not having to deal with common blocks. I also found the forall eval operation to be very helpful. The value oriented nature of the language makes use of the language cleaner. (b) I found input and output to be awkward. For a research language, the run-time information produced by VALSYS was very disappointing.

Question 5: Describe the experience you had in changing from your usual programming techniques to those of data flow.

Briggs (AI):

I had to go back to the algorithm since translation from Prolog to VAL would involve changing the entire structure of the program. As mentioned above, whenever sequential computation was needed, a series of variable definitions were constructed which implicitly carried out the computation. Parallelisms not exploited in the Prolog were made explicit in VAL. The lack of side effects required a different approach since the Prolog program outputs at the bottom level or the leaves of the computation tree, in VAL the output had to be done at the end. This also I find annoying since the brain obviously does not work this way. The lack of a global database required passing databases as arguments, rather than simply accessing them as in Prolog. The shortcomings of lack of shared memory are commented upon in my report.

Eberhardt (CFD):

The language itself helped change the programming technique since it allows you to follow a logical approach. However, this benefit could be due to lack of experience with the language and the fact that no machine will run compiled VAL. Therefore, it is difficult to assess the changes in programming technique because many of the changes could be due to lack of knowledge.

Levin (Chemistry):

The two outstanding observations were: (1) The "number crunching" implementation was very easy, but (2) the data handling and I/O control was almost totally lacking, or, at least, not yet defined.

Merriam (CFD):

No change appears to be required.

Partridge (Chem.):

Programming technique, not surprisingly, did not change -- the workshop was only two weeks.

Patrick (Lin. Sys.):

The application I was considering did not involve converting a large FORTRAN program to VAL. I really only needed to use a few of the constructs so I didn't have any real difficulty in writing the VAL code. My experience with PISCES had already taught me to think "parallel" so that made using VAL less of a problem.

Question 6: How would you foresee that code development for new applications would proceed using VAL versus your usual programming language(s)? Include the effects of the programming environment for each.

Briggs (AI):

The AI community would never accept VAL since they are used to high level programming environments. If such a language was built on top of VAL, and if shared memory was allowed VAL might catch on.

Eberhardt (CFD):

If the compiler can live up to all its promises then I think that code development would be simple. To program without memory management headaches or having to worry about pipelines would be wonderful. It appears that the ideal implementation of data flow and VAL would allow a relatively naive approach to programming. In this study we worked out in more detail how the machine would be "filled" but I think in practice we would not have to.

Levin (Chemistry):

Coding of arithmetic operations is much easier in VAL, but there is a lot missing with regard to data manipulation and also some very optimistic assumptions about what the compiler can do.

Merriam (CFD):

Assuming that the missing constructs are not required, VAL would be about even with Vectoral. There would of course be a large training and code conversion expense.

Partridge (Chem.):

The emphasis on vectorizing inner loops would be relaxed. This would eliminate some of the contortions one goes through. By the same token however, similar rearrangements might be needed to localize memory references.

Patrick (Lin. Sys.):

I feel I would need to do a really large application program and get comfortable with VAL before I could make a reasonable comparison. Given my limited experience with VAL, my impression is that VAL would be a clean, nice language to use for a large code. I got no real experience with I/O in VAL but I have the impression it may be a problem.

Question 7: Do you think the data flow techniques presented and practiced in this study are or will become useful for your applications? Are they more "natural" for your problem domain than conventional concepts?

Briggs (AI):

The basic concept of parallelism is more natural than pure sequentialism. I am not convinced that data flow models are more natural however. Certainly, shared memory is more natural than lack thereof: and while some computations in the brain are "data-driven", not all are.

Eberhardt (CFD):

There is no doubt that the data flow concepts are more natural to my CFD applications. The architecture allows matrix elements to be computed concurrently as though defining the matrix at once, instead of defining the elements sequentially. I think that if the machine works, there will be a large number of CFD users (except the conservative ones).

Levin (Chemistry):

Looks ok, but no obvious advantages noted.

Merriam (CFD):

NO. NO.

Partridge (Chem.):

I think there is sufficient parallelism present in our application that if such a machine is built that we will be able to use it.

Patrick (Lin. Sys.):

See answer to the question immediately preceding this one.

Question 8: Does data flow allow or encourage you to consider your approach to problem solving in your applications area in a new light? If so, did this lead to new ways of thinking about your application?

Briggs (AI):

Data flow did encourage me to think of AI problems in a new light, but it was the concept of parallelism that was the crucial difference in my thinking. Other aspects of data flow did not seem applicable to the domain of AI.

Eberhardt (CFD):

The primary insight I got into my algorithm from the study was a method for performing Gaussian Elimination across an array of processors. The processors do not necessarily have to be data flow either. With data flow, however, I feel that less will be required of me in terms of extracting parallelism.

Levin (Chemistry):

Not particularly.

Merriam (CFD):

No. No more than any extended look at an algorithm.

Partridge (Chem.):

No, but the algorithm (or my implementation) will probably need to be modified to be efficiently mapped.

Patrick (Lin. Sys.):

Again experience gained from this study is limited. However, I felt, when thinking about the VAL implementation of my problem solution, the need to write code that could be pipelined. Also I was concerned as to whether my code had been written so that computation and communication could be overlapped. Without, this my code will perform very poorly on the static data flow machine. I am interested to know whether other participants felt the same pressures concerning pipelining and communication and computation overlap when writing their code.

Question 9: What percent of the way did you get toward preparing you application for data flow during the study?

Briggs (AI):

The program was completely rewritten.

Eberhardt (CFD):

My CSCM code in one-dimension is completed except the I/O and initialization. I intend to complete it and test it on the interpreter if possible. I would also like to see it run completely (to convergence) on a prototype machine, if possible. If the prototype test is successful, I would be willing go program a two-dimensional or three-dimensional problem for future study.

Levin (Chemistry):

I finished a small but computationally intensive module that is 10 percent of the total code.

Merriam (CFD):

The jury isn't in yet but I would say about 80%

Partridge (Chem.):

?

Patrick (Lin. Sys.):

As mentioned earlier, my code was relatively simple so I was able to complete the code.

Question 10: What do you think are the strong and weak points of the static data flow architecture?

Briggs (AI):

The basic weakness is lack of shared memory.

Eberhardt (CFD):

The strong points of the static architecture is that less network routing is needed. However, there is a problem if you have a reasonably long and wide pipeline. You may have to spread the pipe over several processors since each processors instruction memory is not large enough. Also, in my application, certain functions are executed at different times. In particular, there is the block tridiagonal fill routines and the block tridiagonal inversion routines. Both of these blocks of code must reside in the processors instruction memory even though they never need to overlap in some applications. It may make sense to allow them to be allocated to processors dynamically.

Levin (Chemistry):

Strong: Easy to get massive parallelism. Weak: Doesn't work too well for short pipelines. Also, may have trouble with routing network for problems that require lots of data to be sent between PEs.

Merriam (CFD):

The strong points are as follows: (a) better resolution of memory contention problems in multiprocessing; (b) potential for better performance per unit cost through use of lower technology chips; (c) better treatment of exceptions (boundary conditions) and indirect addressing; (d) hardware support for structured data types; and (e) potential for better scaling of performance to machine complexity. The weak points are: (a) only large problems can run fast and (b) a very difficult compiler problem. Further difficulties in the suggested hardware implementation are; (a) a severe bottleneck in the Array Memory bandwidth; (b) a difficulty in choosing the size of the instruction buffer (too small, PE goes idle; too big, a crucial instruction isn't done. another PE goes idle); (c) no I/O; (d) one user; and (e) a difficult network problem.

Partridge (Chem.):

The strongest point is that it allows one to exploit enormous degrees of parallelism. This can be done even when the code would not easily vectorize. The weak points are: VAL, static code allocation to processors, and I/O.

Patrick (Lin. Sys.):

I want to delay a careful answer to this question until I finish my analysis of how my code maps onto the static data flow machine. The close relationship between the data flow graph for a computation and the architecture of the machine is very appealing. A concern is whether the routing network is really a problem or not (i.e., is it a bottleneck). Also the idea that every instruction must receive an acknowledge signal before it can fire again is bothersome. This seems like synchronization after every operation. My experience to date is that the cost of synchronization is what most often makes parallel computation not cost effective. More importantly, it appears that the key to success is pipelining and the ability of the VAL compiler to translate the code so that data can be pipelined through the instructions in the PEs. An important question is what is the overhead associated with this translation and what is the nature of the computations which allow this cost to be affordable. I feel that this question should be carefully considered. One final issue is the amount of storage required for multiple copies of the instructions and data required to make optimal use of the hardware. I still do not clearly understand the ramifications of the idea that arrays are treated as values and when multiple copies of arrays need to be stored.